

Models of Computation

Computation of an Algorithm

Algorithms are executed on machines — it can be for example:

- real computer — executes instructions of a machine code
- virtual machine — executes instructions of a bytecode
- some idealized mathematical model of a computer
- ...

The machine can be:

- specialized — executes only one algorithm
- universal — can execute arbitrary algorithm, given in a form of **program**

The machine performs **steps**.

The algorithm processes a particular **input** and produces the corresponding **output** during its computation.

Model of Computation — an idealized mathematical model of a computer

- abstracts from some unimportant implementation details
- we want to analyze those properties of algorithms that are as much as possible independent of details of a machine that will execute the given algorithm

Examples of some models of computation:

- finite automata
- pushdown automata
- Turing machines
- random-access machines
- ...

Models of Computation

During a computation, the machine must remember:

- the current instruction
- the content of its working memory

It depends on the type of the machine:

- what is the type of data, with which the machine works
- how this data are organized in its memory
- what kind of operations the machine can do with this data

Depending on the type of the algorithm and the type of analysis, which we want to do, we can decide if it makes sense to include in memory also the places

- from which the input data are read
- where the output data are written

One role, for which models of computations are used for, is to define precisely some notions that are important for specifying **computational complexity** of a given algorithm:

- **running time** of a given algorithm \mathcal{A} for a given input w
(remark: typically, it is a number of steps performed during the computation by the machine)
- **amount of memory** used by the machine during this computation

In general, it is also important for different models of computation

- whether a given type of machine is able to **simulate** computations of some other type of machine
- how the running time or the amount of used memory differs compared to the original machine

Simulation of a Computation

Explanation what it means that a machine \mathcal{M} is **simulated** by a machine \mathcal{M}' :

- A computation of machine \mathcal{M} for input w is a (finite or infinite) sequence of configurations of machine \mathcal{M}

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \dots$$

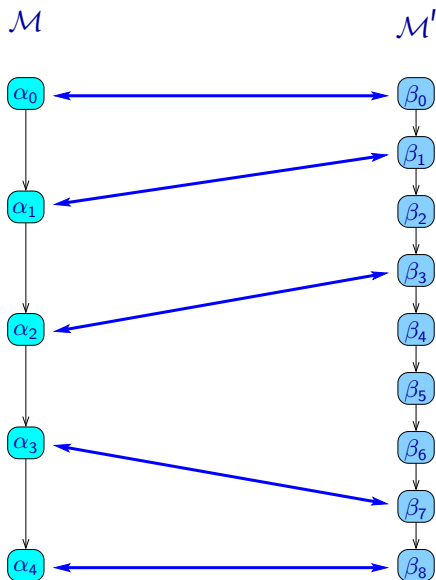
- For this computation, there is a corresponding computation of machine \mathcal{M}' consisting of configurations

$$\beta_0 \longrightarrow \beta_1 \longrightarrow \beta_2 \longrightarrow \dots$$

where for every configuration α_i there is some corresponding configuration $\beta_{f(i)}$ where $f : \mathbb{N} \rightarrow \mathbb{N}$ is a function, for which $f(i) \leq f(j)$ for every i and j where $i < j$.

- There is a relation between configurations of machine \mathcal{M} to configurations of machine \mathcal{M}' that correspond to them.
- There are functions mapping an input w to corresponding initial configurations α_0 and β_0 and analogously functions mapping final configurations to a result of computation.

Simulation of a Computation



Some models of computation are weaker (finite automata, pushdown automata, ...) and they can not be used to implement an arbitrary algorithm.

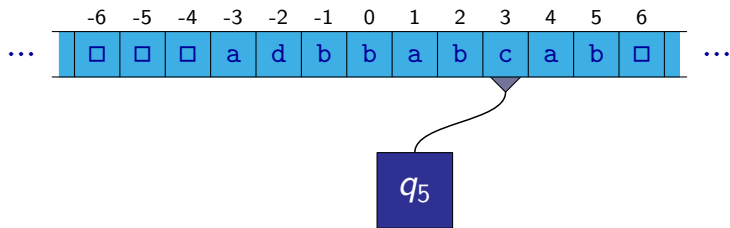
We will concentrate now on models of computation that are powerful enough to be able to execute arbitrary algorithm (for example such that can be represented as a program in some programming language).

Such models of computation are called **Turing-complete**:

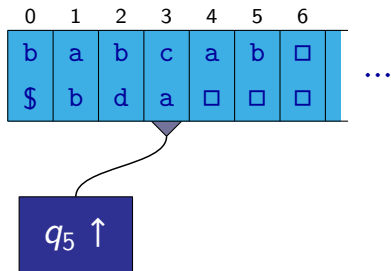
- they are able to simulate a behaviour of arbitrary Turing machine
- their behaviour can be simulated by a Turing machine

Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

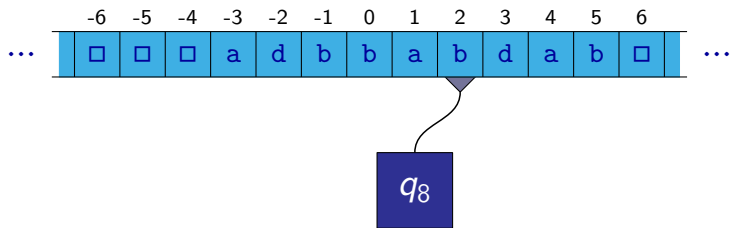


A tape infinite only on one side:

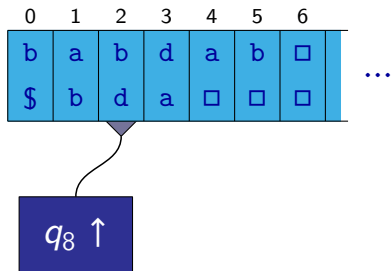


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

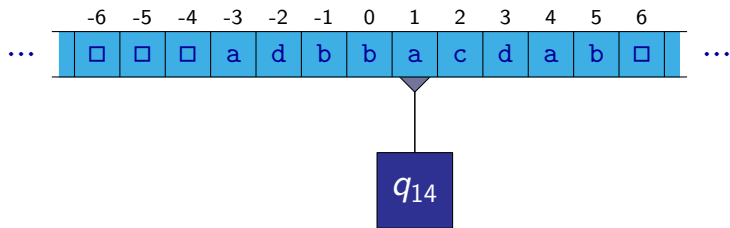


A tape infinite only on one side:

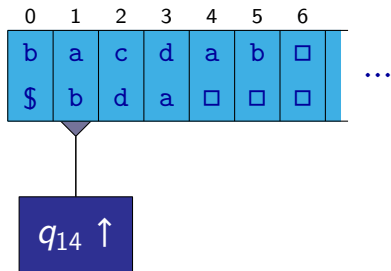


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

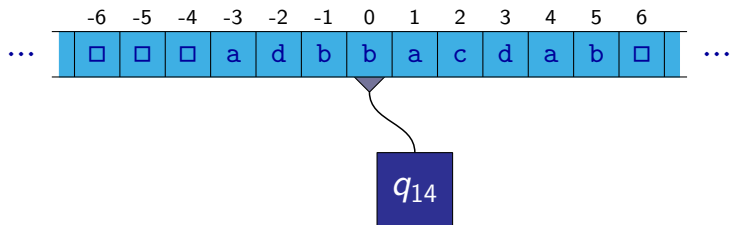


A tape infinite only on one side:

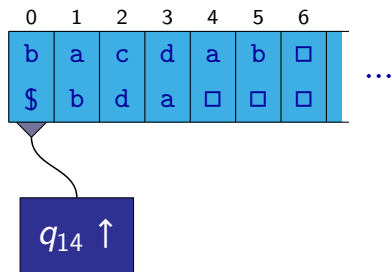


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

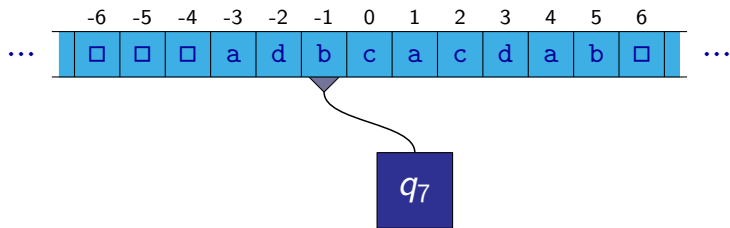


A tape infinite only on one side:

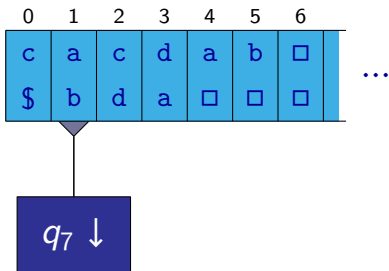


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

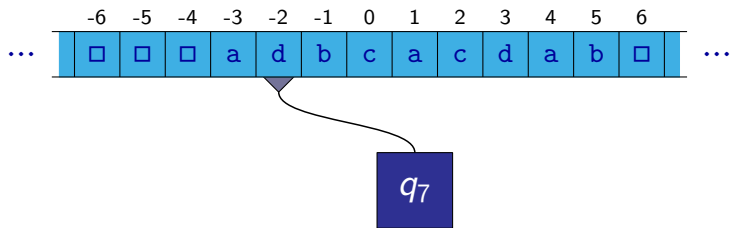


A tape infinite only on one side:

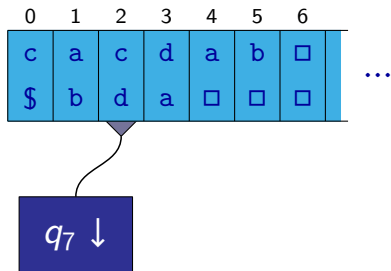


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

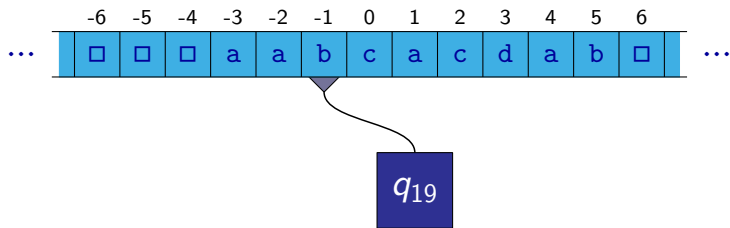


A tape infinite only on one side:

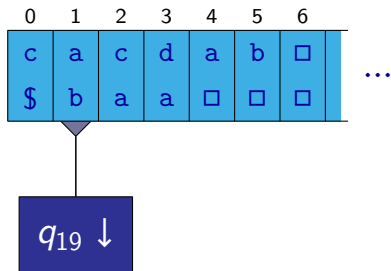


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

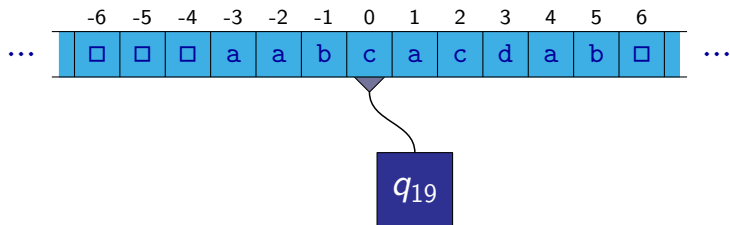


A tape infinite only on one side:

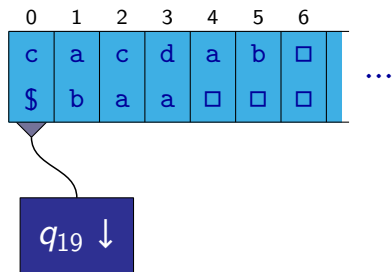


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

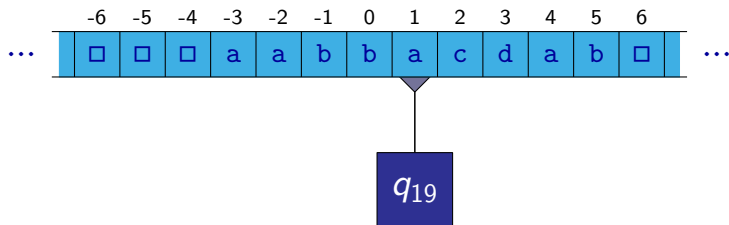


A tape infinite only on one side:

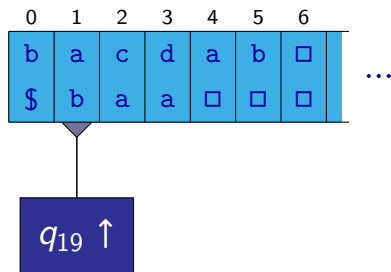


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:

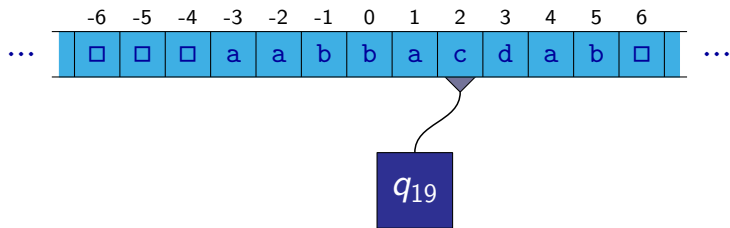


A tape infinite only on one side:

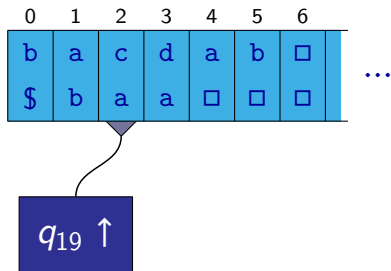


Two-Sided Infinite Tape by One-Sided Infinite Tape

A tape infinite on both sides:



A tape infinite only on one side:



Alphabet $\{0, 1\}$

A Turing machine with an arbitrary tape alphabet Γ can be simulated by a Turing machine with tape alphabet $\{0, 1\}$.

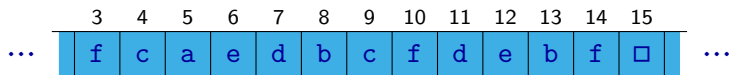
We can choose some appropriate encoding of symbols of alphabet Γ by k -bit sequences.

Example: Tape alphabet $\Gamma = \{\square, a, b, c, d, e, f, g\}$

\square	\leftrightarrow	000
a	\leftrightarrow	001
b	\leftrightarrow	010
c	\leftrightarrow	011
d	\leftrightarrow	100
e	\leftrightarrow	101
f	\leftrightarrow	110
g	\leftrightarrow	111

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

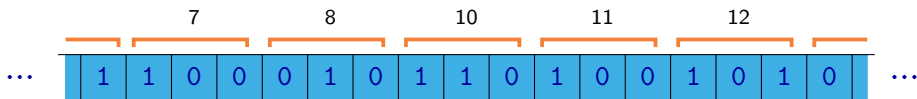


q_7

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

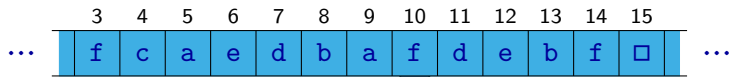
The corresponding machine with alphabet $\{0, 1\}$:



q_7 011

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

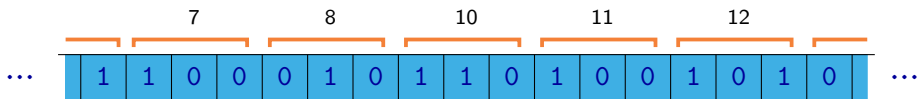


q_{12}

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

The corresponding machine with alphabet $\{0, 1\}$:

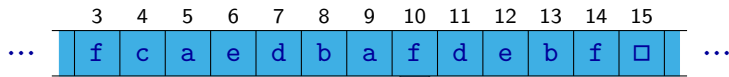


q_{12}

001; ϵ
right

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

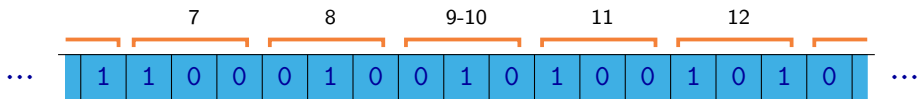


q_{12}

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

The corresponding machine with alphabet $\{0, 1\}$:

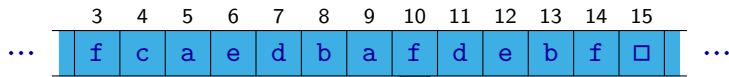


q_{12}

01; 1
right

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

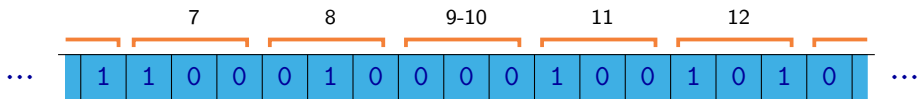


q_{12}

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

The corresponding machine with alphabet $\{0, 1\}$:

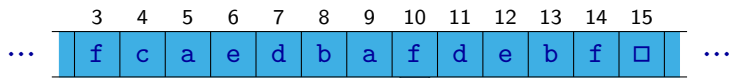


q_{12}

1; 11
right

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

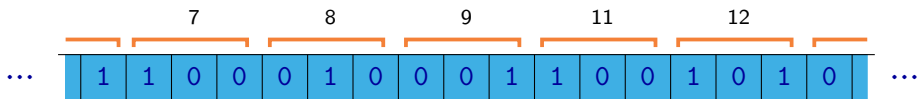


q_{12}

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

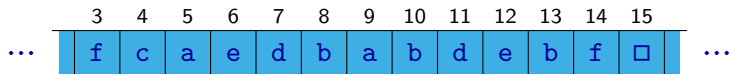
The corresponding machine with alphabet $\{0, 1\}$:



q_{12} 110

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

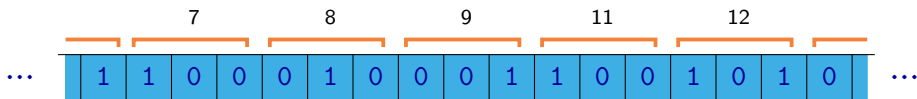


q_5

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

The corresponding machine with alphabet $\{0, 1\}$:

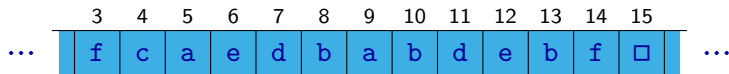


q_5

$\epsilon; 010$
left

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

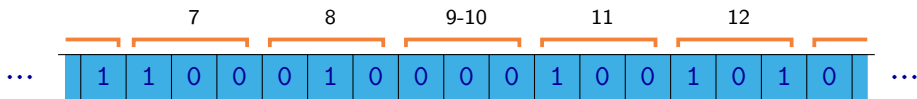


q_5

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

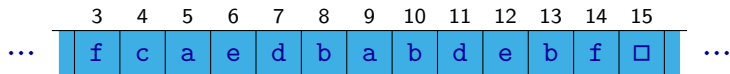
The corresponding machine with alphabet $\{0, 1\}$:



q_5 1; 01
left

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

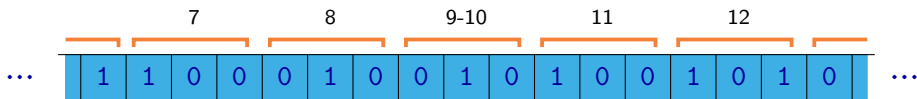


q_5

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

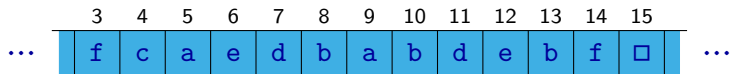
The corresponding machine with alphabet $\{0, 1\}$:



q_5 01; 0
left

Alphabet $\{0, 1\}$

A machine with tape alphabet Γ :

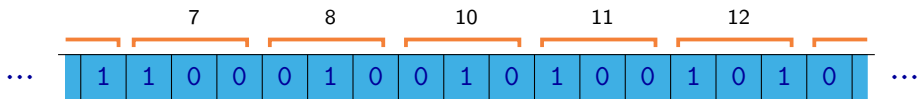


q_5

$$\delta(q_7, c) = (q_{12}, a, +1)$$

$$\delta(q_{12}, f) = (q_5, b, -1)$$

The corresponding machine with alphabet $\{0, 1\}$:



q_5 001

In this simulation, each step of the original machine is simulated by $k + 1$ steps where k is the number of bits used for encoding of one symbol of alphabet Γ .

So if the original machine performs t steps in a computation, the simulating machine performs $O(t)$ steps.

Decreasing the number of states of the control unit

Remark: Similarly, as is possible to decrease the tape alphabet to only two symbols by increasing the number of states of the control unit, it is also possible to decrease the number of states of the control unit:

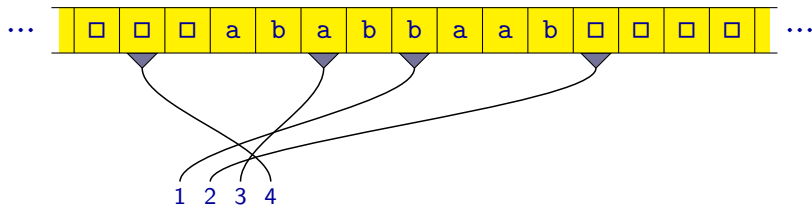
- An arbitrary Turing machine can be simulated by a Turing machine with only two non-final states of its control unit (and possibly with some final states). However, this simulation requires increase in the size of the tape alphabet.

Similarly as in the previous case, one step of the original machine is simulated by s steps where s is a constant depending only on the number of the states of the control unit of the original machines (i.e., the size of set Q).

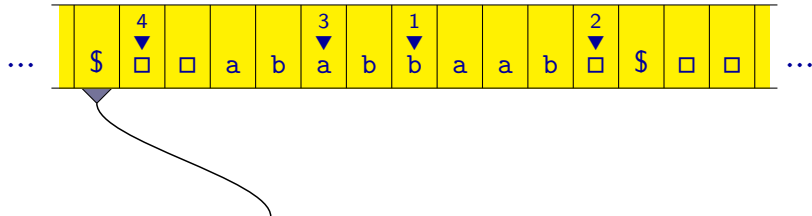
So as before, if the original machine performs t steps, the simulating machine performs $O(t)$ steps.

Simulation of several heads on a tape with one head

Several heads on a tape:

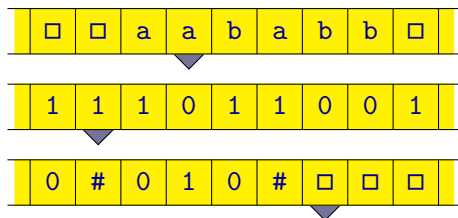


A tape with one head:

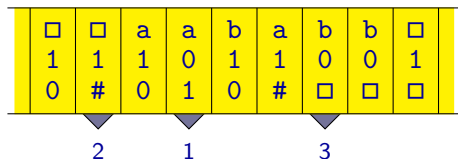


Simulation of several tapes with one tape

Several tapes:

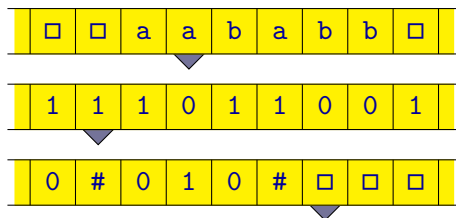


One tape with several heads:

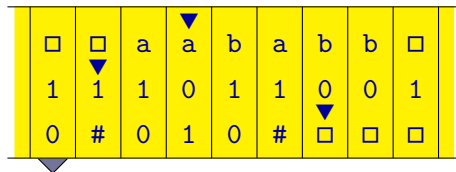


Simulation of several tapes with one tape

Several tapes:

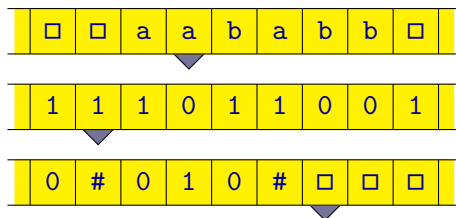


One tape with one head: the variant where where marks on the tape are moved

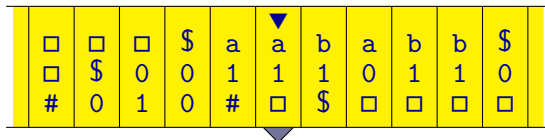


Simulation of several tapes with one tape

Several tapes:



One tape with one head: the variant where the content of tapes is moved



Tapes, stacks, and counters

We consider different types of machines that have a finite control unit equipped with some sort of memory of unbounded size.

Such memory can consist of one or more structures such as:

- **Tape** — reading and writing a symbol on a current position, movement of the head to the left and to the right

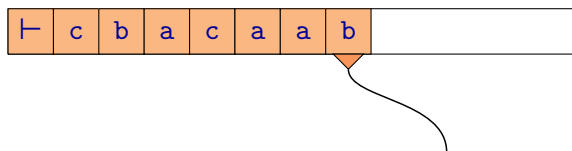
Remark: The tape can be infinite on one side or on both sides.

- **Stack** — push, pop, a test of emptiness of the stack
- **Counter** — a value is a natural number, operations of incrementing and decrementing by one, a test whether the value is equal to zero

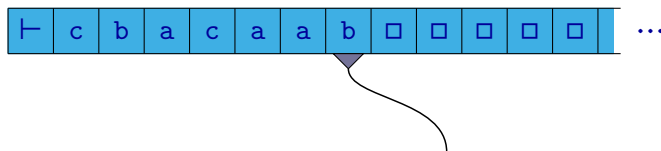
Stack

A stack can be viewed as a special case of a tape, which is infinite on one side.

Stack:



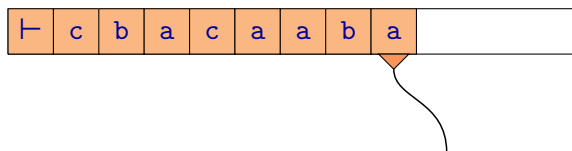
Tape:



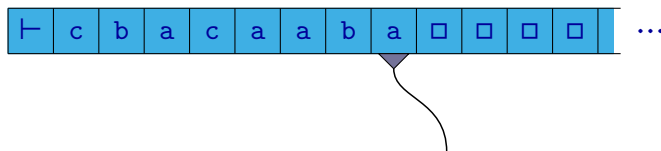
Stack

A stack can be viewed as a special case of a tape, which is infinite on one side.

Stack:



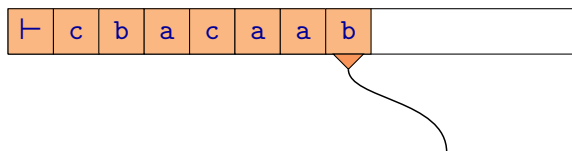
Tape:



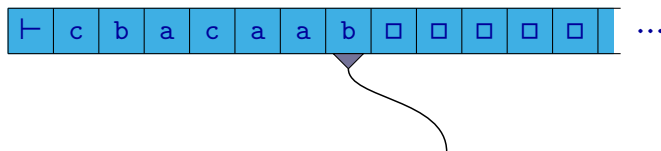
Stack

A stack can be viewed as a special case of a tape, which is infinite on one side.

Stack:



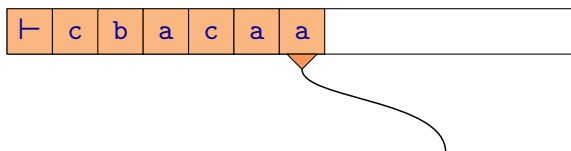
Tape:



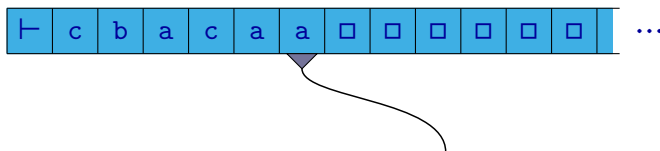
Stack

A stack can be viewed as a special case of a tape, which is infinite on one side.

Stack:



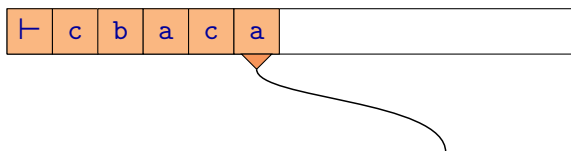
Tape:



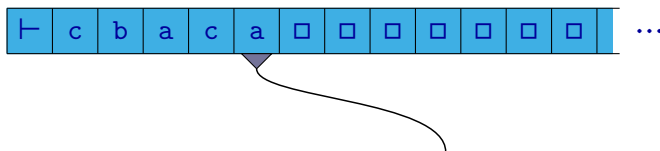
Stack

A stack can be viewed as a special case of a tape, which is infinite on one side.

Stack:

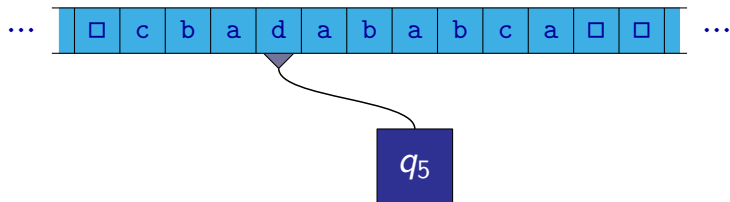


Tape:

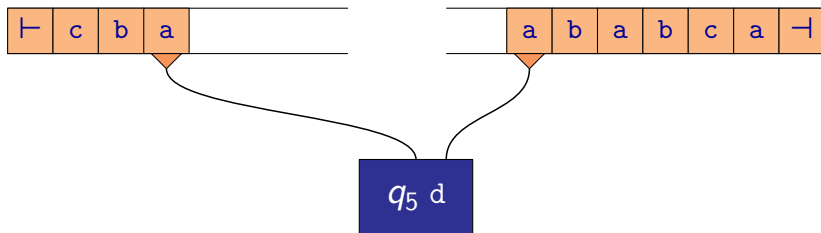


Stack

A tape, infinite on both sides, can be simulated by two stacks:

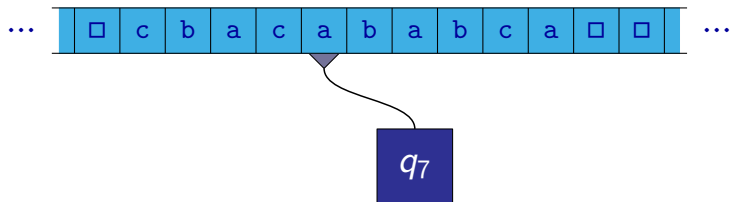


A machine with two stacks:

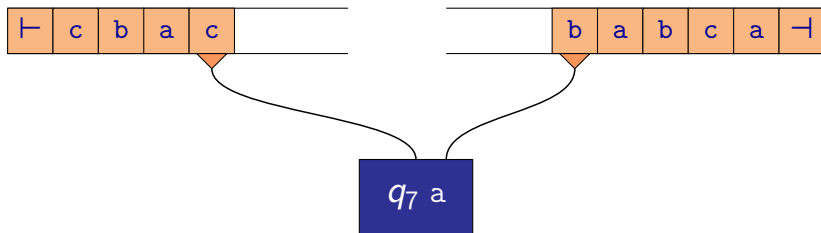


Stack

A tape, infinite on both sides, can be simulated by two stacks:



A machine with two stacks:



Counter — a value of a counter can be an arbitrarily big natural number, i.e., an element of the set $\mathbb{N} = \{0, 1, 2, 3, \dots\}$.

Basic operations:

- incrementing the value by one:

$$x := x + 1$$

- decrementing the value by one:

$$x := x - 1$$

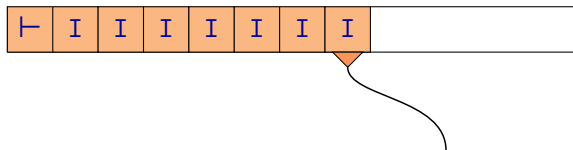
- test whether the value of the counter is zero:

if ($x = 0$) **goto** ℓ

Counter

A counter can be viewed as a special case of a stack or of a tape.

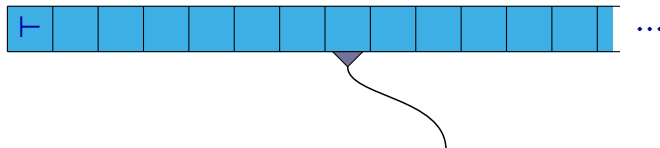
Stack:



Counter:



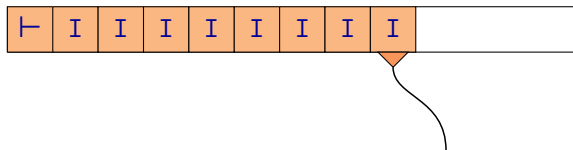
Tape:



Counter

A counter can be viewed as a special case of a stack or of a tape.

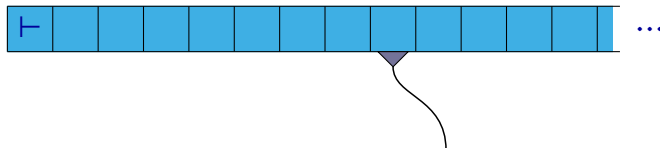
Stack:



Counter:



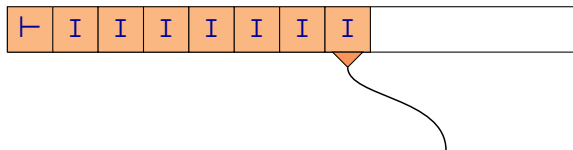
Tape:



Counter

A counter can be viewed as a special case of a stack or of a tape.

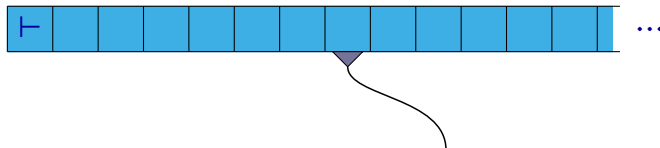
Stack:



Counter:



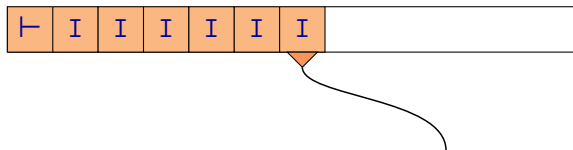
Tape:



Counter

A counter can be viewed as a special case of a stack or of a tape.

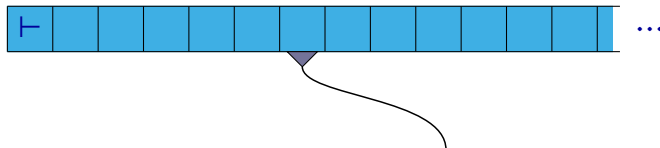
Stack:



Counter:



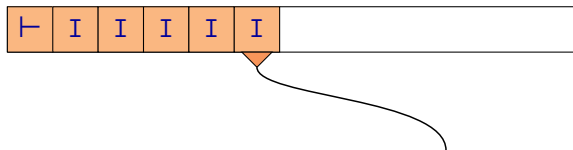
Tape:



Counter

A counter can be viewed as a special case of a stack or of a tape.

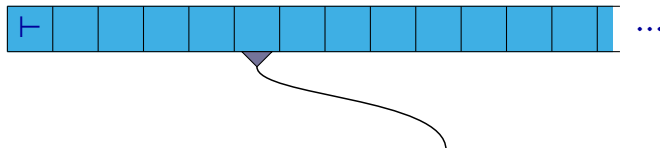
Stack:



Counter:

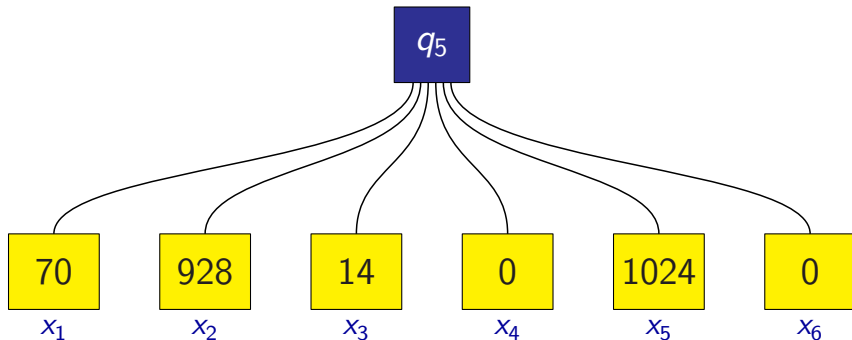


Tape:



Minsky machine

Minsky machine — a machine with a finite control unit and a finite set of counters x_1, x_2, \dots, x_k :



Remark: In addition to symbols x_1, x_2, \dots , we can also use symbols such as x, y, z, \dots to denote counters.

Minsky machine

A Minsky machine can be viewed as a program consisting of a sequence of instructions, with the following five types of instructions:

- incrementing the value of a given counter by one:

$$x_i := x_i + 1$$

- decrementing the value of a given counter by one:

$$x_i := x_i - 1$$

- test whether the value of a given counter is zero:

if ($x_i = 0$) **goto** ℓ

- unconditional jump:

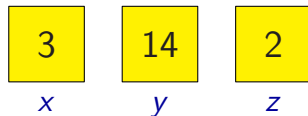
goto ℓ

- halting of the computation of the program:

halt

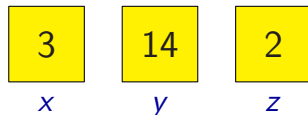
Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



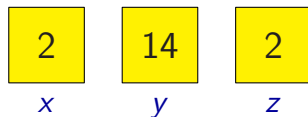

Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



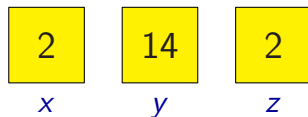
Setting the counter x to zero:

L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



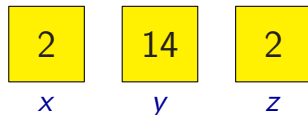
Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



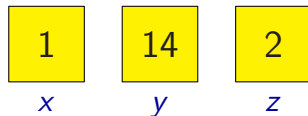
Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



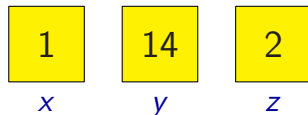
Setting the counter x to zero:

L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
→ **goto** L_1
 L_2 : ...



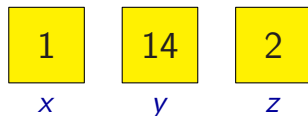
Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



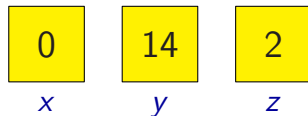
Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



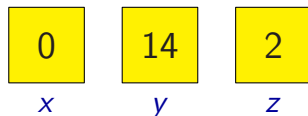
Setting the counter x to zero:

L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



Setting the counter x to zero:

→ L_1 : **if** ($x = 0$) **goto** L_2
 $x := x - 1$
 goto L_1
 L_2 : ...



Setting the counter x to zero:

L_1 : **if** ($x = 0$) **goto** L_2

$x := x - 1$

goto L_1

→ L_2 : ...

0

x

14

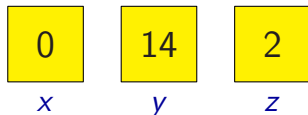
y

2

z

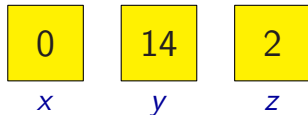
Adding the value of the counter z to the counter y (together with setting the counter z to zero):

→ L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



Adding the value of the counter z to the counter y (together with setting the counter z to zero):

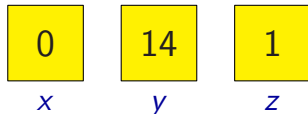
→ L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



Minsky machine

Adding the value of the counter z to the counter y (together with setting the counter z to zero):

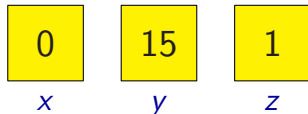
L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



Minsky machine

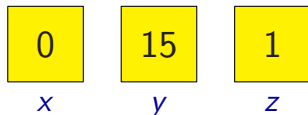
Adding the value of the counter z to the counter y (together with setting the counter z to zero):

```
 $L_2$  : if ( $z = 0$ ) goto  $L_3$   
       $z := z - 1$   
       $y := y + 1$   
      goto  $L_1$   
 $L_3$  : ...
```



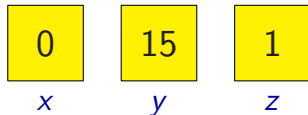
Adding the value of the counter z to the counter y (together with setting the counter z to zero):

→ L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



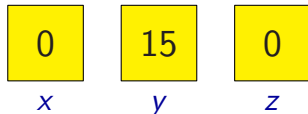
Adding the value of the counter z to the counter y (together with setting the counter z to zero):

→ L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



Adding the value of the counter z to the counter y (together with setting the counter z to zero):

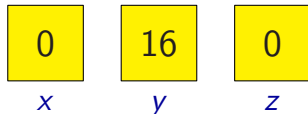
L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...



Minsky machine

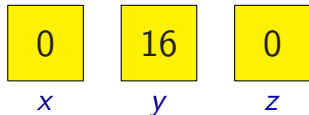
Adding the value of the counter z to the counter y (together with setting the counter z to zero):

```
 $L_2$  : if ( $z = 0$ ) goto  $L_3$   
       $z := z - 1$   
       $y := y + 1$   
      goto  $L_1$   
 $L_3$  : ...
```



Adding the value of the counter z to the counter y (together with setting the counter z to zero):

→ L_2 : **if** ($z = 0$) **goto** L_3
 $z := z - 1$
 $y := y + 1$
 goto L_1
 L_3 : ...

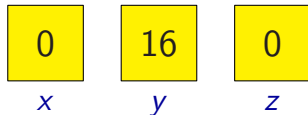


Minsky machine

Adding the value of the counter z to the counter y (together with setting the counter z to zero):

```
 $L_2$  : if ( $z = 0$ ) goto  $L_3$   
       $z := z - 1$   
       $y := y + 1$   
      goto  $L_1$ 
```

→ L_3 : ...



Multiplying the value of counter x with constant 5:

L_1 : **if** ($x = 0$) **goto** L_2

$x := x - 1$

$y := y + 1$

$y := y + 1$

$y := y + 1$

$y := y + 1$

$y := y + 1$

goto L_1

L_2 : **if** ($y = 0$) **goto** L_3

$y := y - 1$

$x := x + 1$

goto L_2

L_3 : ...

Minsky machine

Division of the value of the counter x with constant 5 and finding out the remainder after this division:

```
 $L_1$  : if ( $x = 0$ ) goto  $M_0$   
       $x := x - 1$   
      if ( $x = 0$ ) goto  $M_1$   
       $x := x - 1$   
      if ( $x = 0$ ) goto  $M_2$   
       $x := x - 1$   
      if ( $x = 0$ ) goto  $M_3$   
       $x := x - 1$   
      if ( $x = 0$ ) goto  $M_4$   
       $x := x - 1$   
       $y := y + 1$   
      goto  $L_1$ 
```

A stack can be simulated using a pair of counters — a value of the first counter represents the content of the stack as a number of base $k = |\Gamma| + 1$ (where Γ is a stack alphabet).

- A stack on the top of the stack — the remainder of division by k
- Pop — to divide by k
- Push — to multiply by k and to add the code of the given symbol

The second counter is used as an auxiliary counter for performing the above given operations.

Example:

a \leftrightarrow 1

b \leftrightarrow 2

c \leftrightarrow 3

d \leftrightarrow 4

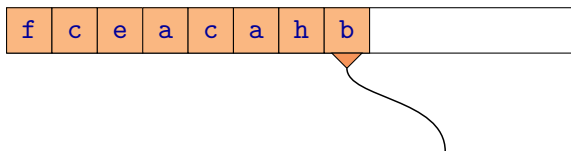
e \leftrightarrow 5

f \leftrightarrow 6

g \leftrightarrow 7

h \leftrightarrow 8

i \leftrightarrow 9

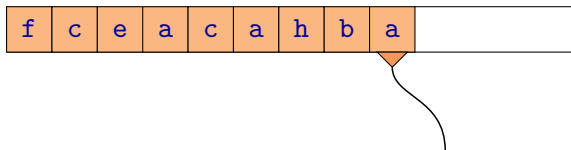


63513182

Minsky machine

Example:

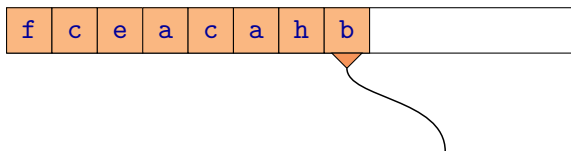
a ↔ 1
b ↔ 2
c ↔ 3
d ↔ 4
e ↔ 5
f ↔ 6
g ↔ 7
h ↔ 8
i ↔ 9



635131821

Example:

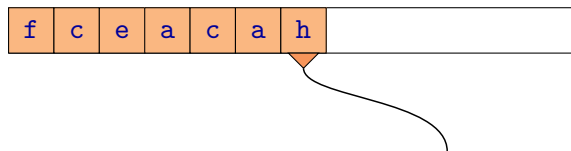
a \leftrightarrow 1
b \leftrightarrow 2
c \leftrightarrow 3
d \leftrightarrow 4
e \leftrightarrow 5
f \leftrightarrow 6
g \leftrightarrow 7
h \leftrightarrow 8
i \leftrightarrow 9



63513182

Example:

a \leftrightarrow 1
b \leftrightarrow 2
c \leftrightarrow 3
d \leftrightarrow 4
e \leftrightarrow 5
f \leftrightarrow 6
g \leftrightarrow 7
h \leftrightarrow 8
i \leftrightarrow 9



6351318

Minsky machine

Example:

a ↔ 1

b ↔ 2

c ↔ 3

d ↔ 4

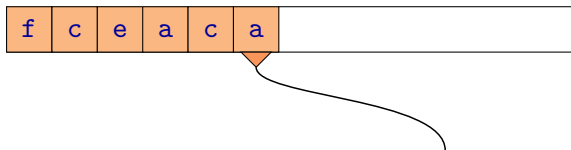
e ↔ 5

f ↔ 6

g ↔ 7

h ↔ 8

i ↔ 9



635131

Recall that a tape infinite of both sides can be simulated by a pair of stacks.

In a Minsky machine, the content of each of these stacks can be represented by a corresponding counter.

We also need one additional counter for the implementation of multiplication and division by a constant on these counters representing contents of the stacks.

We can see that a Turing machine with k tapes can be simulated by a Minsky machine with $2k + 1$ counters.

Any finite number of counters can be simulated by two counters.

- One counter (let it be denoted as C) represents values of all counters — e.g., values of three counters x , y , z can be represented in the counter C by the number $2^x 3^y 5^z$.
- The second counter is used as an auxiliary counter to perform operations of multiplication and division on counter C .
- Incrementing counter x by one is simulated as multiplying by 2, incrementing counter y by one is simulated as multiplying by 3, etc.
- In a similar way, decrementing counter x by one is simulated by division of counter C by number 2, decrementing counter y by one by division by number 3, etc.
- The test if $x = 0$ corresponds to test if the value of counter C is divisible by 2, etc.

We can see that computation of an arbitrary Turing machine can be simulated by a Minsky machine with two counters.

This simulation is extremely inefficient:

- Already simulation of a tape of a Turing machine by three counters requires number of steps that is exponentially bigger than the number of steps performed by this Turing machine.
- Simulation of these three counters using only two counters farther exponentially increases the performed number of steps.

Random Access Machines

Random Access Machine

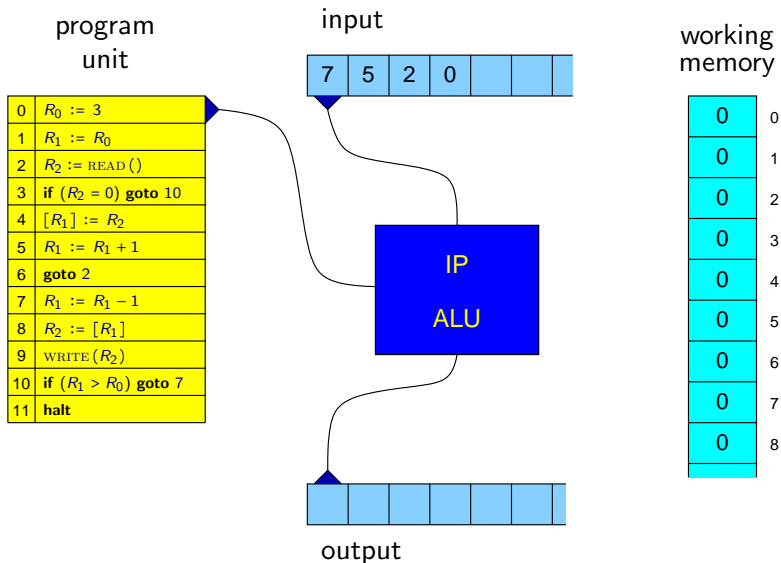
A **Random Access Machine (RAM)** is an idealized model of a computer.

It consists of the following parts:

- **Program unit** – contains a program for the RAM and a pointer to the currently executed instruction
- **Working memory** consists of cells numbered $0, 1, 2, \dots$
These cells will be denoted R_0, R_1, R_2, \dots
The content of the cells can be read and written to.
- **Input tape** – read-only
- **Output tape** – write-only

The cells of memory, as well as the cells of input and output tapes contain integers (i.e., elements of set \mathbb{Z}) as their values.

Random Access Machine



Random Access Machine

Overview of instructions:

- $R_i := c$ – assignment of a constant
- $R_i := R_j$ – assignment
- $R_i := [R_j]$ – load (reading from memory)
- $[R_i] := R_j$ – store (writing to memory)
- $R_i := R_j \text{ op } R_k$ – arithmetic instructions, $op \in \{+, -, *, /\}$
or $R_i := R_j \text{ op } c$
- if** ($R_i \text{ rel } R_j$) **goto** ℓ – conditional jump, $rel \in \{=, \neq, \leq, \geq, <, >\}$
or **if** ($R_i \text{ rel } c$) **goto** ℓ
- goto** ℓ – unconditional jump
- $R_i := \text{READ}()$ – reading from input
- $\text{WRITE}(R_i)$ – writing to output
- halt** – program termination

Random Access Machine

Examples of instructions:

$R_5 := 42$

$R_{12} := R_3$

$R_8 := [R_2]$

$[R_{15}] := R_9$

$R_7 := R_3 + R_6$

$R_{18} := R_{18} - 1$

if ($R_4 \geq R_1$) **goto** 2801

if ($R_2 \neq 0$) **goto** 3581

goto 537

$R_{23} := \text{READ}()$

$\text{WRITE}(R_{17})$

halt

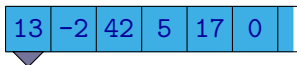
- assignment of a constant
- assignment
- load (reading from memory)
- store (writing to memory)
- arithmetic instruction
- arithmetic instruction
- conditional jump
- conditional jump
- unconditional jump
- reading from input
- writing to output
- program termination

Random Access Machine



```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



Output

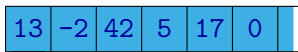
0	?
1	?
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine



```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



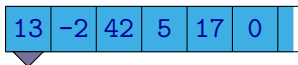
Output

0	3
1	?
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



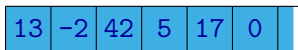
Output

0	3
1	3
2	?
3	?
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
    $\text{WRITE}(R_2)$   
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
   halt
```

Input



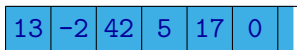
Output

0	3
1	3
2	13
3	?
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
→    $[R_1] := R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```

Input



Output

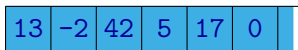
0	3
1	3
2	13
3	?
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
     $[R_1] := R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```



Input



Output

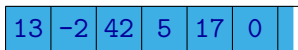
0	3
1	3
2	13
3	13
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```



Input



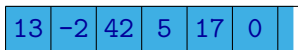
Output

0	3
1	4
2	13
3	13
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



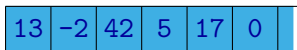
Output

0	3
1	4
2	13
3	13
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
    $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
      halt
```

Input



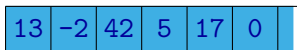
Output

0	3
1	4
2	-2
3	13
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
       $R_2 := [R_1]$   
      WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
      halt
```

Input



Output

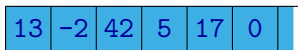
0	3
1	4
2	-2
3	13
4	?
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
    [ $R_1$ ] :=  $R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```



Input



Output

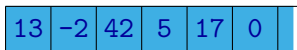
0	3
1	4
2	-2
3	13
4	-2
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```



Input



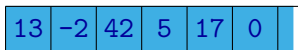
Output

0	3
1	5
2	-2
3	13
4	-2
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



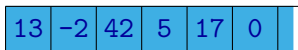
Output

0	3
1	5
2	-2
3	13
4	-2
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



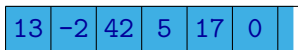
Output

0	3
1	5
2	42
3	13
4	-2
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
→    $[R_1] := R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```

Input



Output

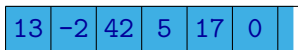
0	3
1	5
2	42
3	13
4	-2
5	?
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
    [ $R_1$ ] :=  $R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```



Input



Output

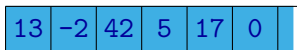
0	3
1	5
2	42
3	13
4	-2
5	42
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```



Input



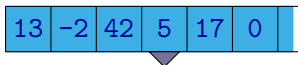
Output

0	3
1	6
2	42
3	13
4	-2
5	42
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



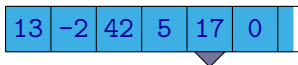
Output

0	3
1	6
2	42
3	13
4	-2
5	42
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
    $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
      halt
```

Input



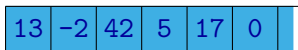
Output

0	3
1	6
2	5
3	13
4	-2
5	42
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
→    $[R_1] := R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```

Input



Output

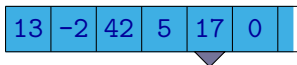
0	3
1	6
2	5
3	13
4	-2
5	42
6	?
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
    if ( $R_2 = 0$ ) goto  $L_3$   
    [ $R_1$ ] :=  $R_2$   
     $R_1 := R_1 + 1$   
    goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
     $R_2 := [R_1]$   
    WRITE( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
    halt
```



Input



Output

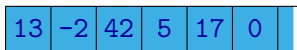
0	3
1	6
2	5
3	13
4	-2
5	42
6	5
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```



Input



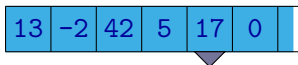
Output

0	3
1	7
2	5
3	13
4	-2
5	42
6	5
7	?
8	?
9	?
10	?
11	?

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



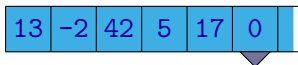
Output

0	3
1	7
2	5
3	13
4	-2
5	42
6	5
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
   WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
   halt
```

Input



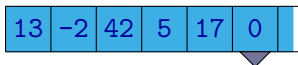
Output

0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
   WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
   halt
```

Input



Output

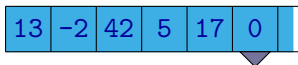
0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	?
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```



Input



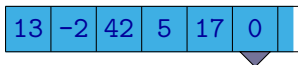
Output

0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	8
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

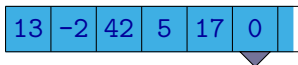


Output

Random Access Machine

$R_0 := 3$
 $R_1 := R_0$
→ $L_1 : R_2 := \text{READ}()$
 if ($R_2 = 0$) **goto** L_3
 $[R_1] := R_2$
 $R_1 := R_1 + 1$
 goto L_1
 $L_2 : R_1 := R_1 - 1$
 $R_2 := [R_1]$
 WRITE(R_2)
 $L_3 : \text{if}$ ($R_1 > R_0$) **goto** L_2
 halt

Input



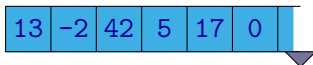
Output

0	3
1	8
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
→ if ( $R_2 = 0$ ) goto  $L_3$   
    $[R_1] := R_2$   
    $R_1 := R_1 + 1$   
   goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
   WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
   halt
```

Input



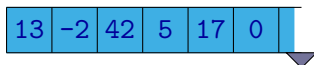
Output

0	3
1	8
2	0
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



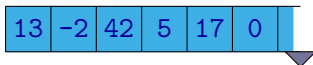
Output

0	3
1	8
2	0
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
→  $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
    $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



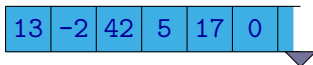
Output

0	3
1	8
2	0
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	7
2	0
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

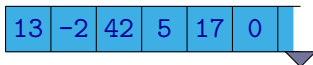


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

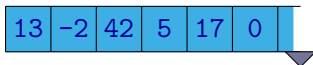


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



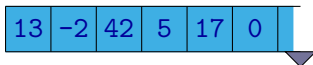
Output

0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
→  $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	7
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?



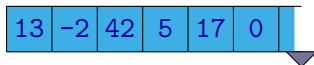
Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```



Input



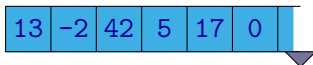
Output

0	3
1	6
2	17
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	6
2	5
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

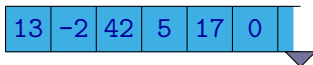


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



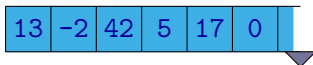
Output

0	3
1	6
2	5
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
→  $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



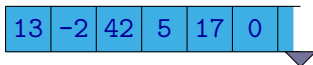
Output

0	3
1	6
2	5
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	5
2	5
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

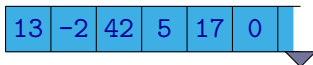


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	5
2	42
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

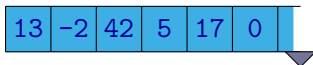


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



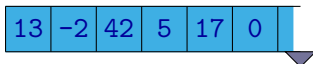
Output

0	3
1	5
2	42
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
→  $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



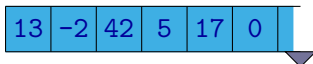
Output

0	3
1	5
2	42
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	4
2	42
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

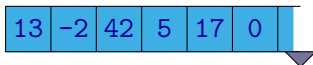


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



0	3
1	4
2	-2
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

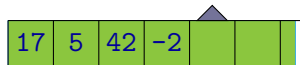
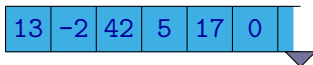


Output

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



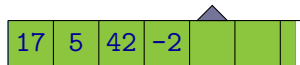
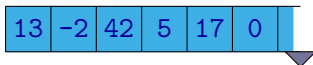
Output

0	3
1	4
2	-2
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
→  $L_2 : R_1 := R_1 - 1$   
    $R_2 := [R_1]$   
    $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



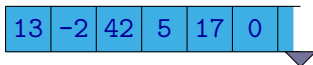
Output

0	3
1	4
2	-2
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

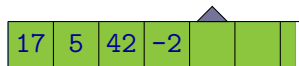
Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	3
2	-2
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

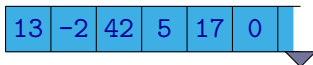


Output

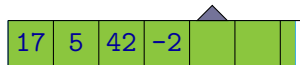
Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
  WRITE ( $R_2$ )  
 $L_3 : \text{if}$  ( $R_1 > R_0$ ) goto  $L_2$   
  halt
```

Input



0	3
1	3
2	13
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

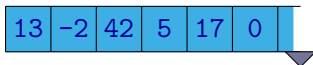


Output

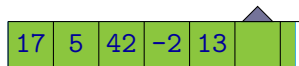
Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
→  $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
  halt
```

Input



0	3
1	3
2	13
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?

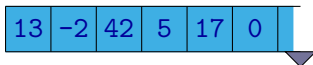


Output

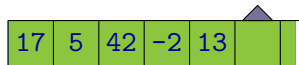
Random Access Machine

```
 $R_0 := 3$   
 $R_1 := R_0$   
 $L_1 : R_2 := \text{READ}()$   
  if ( $R_2 = 0$ ) goto  $L_3$   
   $[R_1] := R_2$   
   $R_1 := R_1 + 1$   
  goto  $L_1$   
 $L_2 : R_1 := R_1 - 1$   
   $R_2 := [R_1]$   
   $\text{WRITE}(R_2)$   
 $L_3 : \text{if } (R_1 > R_0) \text{ goto } L_2$   
halt
```

Input



0	3
1	3
2	13
3	13
4	-2
5	42
6	5
7	17
8	?
9	?
10	?
11	?



Output

Main differences with respect to real computers:

- The size of memory is not limited (an address can be an arbitrary natural number).
- The size of a content of individual memory cells is not limited (a cell can contain an arbitrary integer).
- It reads data sequentially from an input that consists of a sequence of integers. The input is read-only.
- It writes data sequentially on the output that consists of a sequence of integers. The output is write-only.

Random Access Machine

- Operations like an access to a memory cell with an address less than zero or division by zero result in an error — the computation is stuck.
- For an initial content of memory there are basically two possibilities how to define it:
 - All cells are initialized with value 0.
 - Reading a cell, to which nothing has been written, results in an error. Cells at the beginning contain a special value (denoted here by symbol '?') that represents that the given cell has not been initialized yet.
- We could consider also variants of RAMs where memory cells (and cells of input and output) do not contain integers (i.e., the elements of set \mathbb{Z}) but they can contain only natural numbers (i.e., elements of set \mathbb{N}).

For example, operation of subtraction ($R_i := R_j - R_k$) then behaves in such a way that whenever the result should be a negative number, then value 0 is assigned as the result.

Random Access Machine

- Different variants of RAMs can differ in what particular operations can be used in arithmetic instructions.

For example:

- a support of bitwise operations (and, or, not, xor, ...), bit shifts, ...
 - a variant of RAM that does not have operations for multiplication and division
- We could also consider a variant of RAM where instead of instructions of the form

if ($R_i \text{ rel } R_j$) **goto** ℓ nebo **if** ($R_i \text{ rel } c$) **goto** ℓ

all conditional jumps are of the form

if ($R_i \text{ rel } 0$) **goto** ℓ

Instead of all relations $\{=, \neq, \leq, \geq, <, >\}$, only a subset of them can be supported, e.g., $\{=, >\}$.

- In some variants of RAM, the input and output are not in a form of sequence of numbers.

Instead, such machine could work with input and output tapes containing sequences of symbols from some alphabet, e.g., $\{0, 1\}$.

This machine then could have for example some instructions that allow the branch the computation according to a symbol read from the input.

However, the internal memory even in this variant works with numbers.

- When a machine should produce an answer of the form Yes/No (i.e., to accept or reject the given input), it does not need to have an output tape.

Instruction **halt** is then replaced with instructions **accept** and **reject**.

- In the standard definition of RAM, jump instructions jumping to an address stored in some memory cell are usually not considered:

goto R_i

RAM could be extended with these instructions.

- For RAMs, a code of a program is usually stored in a separate read-only memory, not in a working memory.
So the code can not be modified during a computation.

- A type of a machine, similar to RAM, but where its program is stored in its working memory (instructions are encoded by numbers) and so it can be modified during a computations, is called **RASP** (**random-access stored program**).

RASP can simulate behaviour of self-modifying programs.

Turing Machine Simulating RAM

It is not difficult to come with a general way how a computation of an arbitrary Turing machine can be simulated by RAM.

To simulate behaviour of an arbitrary RAM by a Turing machine is more complicated.

In the description of how a Turing machine can simulate a RAM, it is simpler to proceed by smaller steps:

- We will show how to simulate a variant of RAM described before by a variant of RAM with somewhat simpler instructions.
- We will show how to simulate the behaviour of this simpler variant of RAM by a multitape Turing machine.
- We have already seen before how a multitape Turing machine can be simulated by one-tape Turing machine.

A simpler variant of RAM

This simpler variant of RAM has, in addition to its working memory, also three **registers**:

- **register A** — almost all instructions work with this register, results of all operations are stored into this register

Remark: This kind of register is often called an **accumulator**.

- **register B** — this register is used to store the second operand of arithmetic instructions (the first operand is always in the accumulator)
- **register C** — this register is used to store an address of a memory cell, to which a value is written by a store operation

A simpler variant of RAM

Overview of instructions:

$A := c$	– assignment of a constant
$B := A$	– assignment to register B
$C := A$	– assignment to register C
$A := [A]$	– load (reading from memory)
$[C] := A$	– store (writing to memory)
$A := A \text{ op } B$	– arithmetic instructions, $op \in \{+, -, *, /\}$
if ($A \text{ rel } 0$) goto ℓ	– conditional jump, $rel \in \{=, \neq, \leq, \geq, <, >\}$
goto ℓ	– unconditional jump
$A := \text{READ}()$	– reading from input
$\text{WRITE}(A)$	– writing to output
halt	– program termination

A simpler variant of RAM

For example, instruction

$$R_7 := R_3 + R_6$$

can be replaced with a sequence of instructions:

$$A := 7$$

$$C := A$$

$$A := 6$$

$$A := [A]$$

$$B := A$$

$$A := 3$$

$$A := [A]$$

$$A := A + B$$

$$[C] := A$$

A simpler variant of RAM

For example, instruction

$$[R_{15}] := R_9$$

can be replaced with a sequence of instructions:

$$A := 15$$

$$A := [A]$$

$$C := A$$

$$A := 9$$

$$A := [A]$$

$$[C] := A$$

A simpler variant of RAM

For example, instruction

```
if ( $R_4 \geq R_{11}$ ) goto  $\ell$ 
```

can be replaced with a sequence of instructions:

```
A := 11
```

```
A := [A]
```

```
B := A
```

```
A := 4
```

```
A := [A]
```

```
A := A - B
```

```
if ( $A \geq 0$ ) goto  $\ell$ 
```

Turing Machine Simulating RAM

A Turing machine works with words over some alphabet, while a RAM works with numbers. But numbers can be written as sequences of symbols and conversely symbols of an alphabet can be written as numbers.

For example the following input of a RAM

5	13	-3	0	6	
---	----	----	---	---	--

can be represented for a Turing machine as

#	1	0	1	#	1	1	0	1	#	-	1	1	#	0	#	1	1	0	#
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A Turing machine simulating a computation of a RAM has several tapes:

- A tape containing a content of the working memory of the RAM.
- Three tapes containing values of registers A , B , and C .
(Values of registers A , B , and C will be written on these tapes in binary and delimited from the left and from the right by symbols $\#$.)
- A tape representing the input tape of the RAM.
- A tape representing the output tape of the RAM.
- One auxiliary tape used for an implementation of the simulation of some instructions.

Turing Machine Simulating RAM

The Turing machine stores the information about the instruction of the RAM that is currently executed in its control unit.

Execution of most of instructions is not difficult:

- $A := c$
it writes bits of the constant c to the tape of register A
- $B := A$ or $C := A$
it will copy a content of the tape of register A to the tape of register B or C
- **goto** l
just changes the state of the control unit of the Turing machine
- **if** ($A \text{ rel } 0$) **goto** l , kde $\text{rel} \in \{=, \neq, \leq, \geq, <, >\}$
the content of the working register is tested and the state of the control unit is changed accordingly

Turing Machine Simulating RAM

- $A := \text{READ}()$

copy the value (marked at the ends by symbols “#”) from the input tape to the tape of register A

- $\text{WRITE}(A)$

copy the value of register A to the output tape.

- **halt**

the computation halts

Also arithmetic instructions are rather easy to implement, although the a little bit more complicated than the previous instructions:

- $A := A \text{ op } B$, where $\text{op} \in \{+, -, *, /\}$

The Turing machine performs the given operation (such as addition or subtraction) bit by bit, the result is stored to register A .

Remark: Multiplication and division can be done as a sequence of additions and bit shifts.

In the implementation of addition and division, it may be necessary to use an auxiliary tape to store intermediate results.

Turing Machine Simulating RAM

Probably the most complex is the implementation of the RAM memory. One possibility is to store only values of those cells that were actually used so far in the computation of the RAM (we know that all other cells contain value 0).

Example: The RAM worked so far only with cells 2, 3 and 6:

- Cell 2 contains value 11.
- Cell 3 contains value -1.
- Cell 6 contains value 2.

The content of the tape of the Turing machine representing the content of the memory of the RAM will be as follows:

\$	#	1	0	:	1	0	1	1	#	1	1	:	-	1	#	1	1	0	:	1	0	#	\$
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

Load instruction, i.e., $A := [A]$:

- The Turing machine will search the given address, stored in register A , on the tape containing the content of the memory of the RAM. (If it does not find it, it will append it at the end with value 0.)
- The given value in the cell is copied to the tape of register A .

Turing Machine Simulating RAM

Store instruction, i.e., $[C] := A$:

- Similarly as before, the Turing machine will find the position of the tape representing a content of the memory, where the value in the given address, stored in register C , occurs.
- The rest of the memory tape is copied to an auxiliary tape.
- The content of the tape of register A is copied to the corresponding place.
- The rest of the tape, copied on the auxiliary tape, is copied back to the memory tape (after the newly written value).