The Church-Turing thesis states that the Turing Machine is as capable as any realistic model of computation. So we would expect it to perform the same tasks that a real computer can do.

In real computer architectures, the control component is embedded in the central processing unit, or CPU. The computer has an *instruction memory* that stores the program being executed (written in machine language) and *data memory*, which is a vast array of data. In addition to these memories, the CPU contains a small number of *registers* to store values. One of these registers, the *program counter* is special: It is used to record the current line of the program.

In its simplest form, the computer operates as follows. The execution of a program proceeds in uniform time increments, which are signaled by an external device called the *clock*. Before every clock tick, the CPU looks at the program counter and loads the current instruction from instruction memory. A special part of the CPU, called the *arithmetic and logical unit*, or ALU, is in charge of executing the instruction. As the clock ticks, the instruction is executed. This execution may affect the contents of various registers or some slot in the memory. Typically, after the instruction is executed, the program counter is incremented to point to the next instruction in the program, although there are special instructions that can change the course of the program.

Real computer architectures are more complicated than this. They also have to accommodate external devices, such as keyboards, hard drives, screens, and so on. Hardware engineers also attempt to optimize efficiency via more sophisticated designs. Various tricks, such as pipelining, are used to leverage the fact that some instructions take longer time to execute than others. Some computers have parallel processors which allow them to execute several instructions at a time. These are all improvements that play an important role in the practice of computer design. However, as a first approximation, the model we describe here is a fair representation of how computers work.

1 Random access machines

Our model of a computer will be the *random access machine*. This machine consists of the following components:

- A finite (small) number of *registers* $R_0, R_1, \ldots, R_{m-1}, m \geq 1$, which take integer values.
- A *program counter* PC, which takes a non-negative integer value.
- A *memory* M: This is an infinite array $M(0), M(1), \ldots$, where each memory cell $M(k)$ takes an integer value.
- A *program* P: This is a finite sequence of *instructions* described in Table 1.

instruction	meaning	description
`load` n	$R_0 := n$	Put the value n into R_0
`load R`k	$R_0 := R_k$	Copy the value of R_k into R_0
`store R`k	$R_k := R_0$	Copy the value of R_0 into R_k
`read R`k	$R_0 := M[R_k]$	Copy the value at memory location R_k into R_0
`write R`k	$M[R_k] := R_0$	Copy the value of R_0 into memory location R_k
`add` n	$R_0 := R_0 + n$	Add the value n to R_0
`add R`k	$R_0 := R_0 + R_k$	Add the value of R_k into R_0
`mult R`k	$R_0 := R_0 \times R_k$	Multiply the value of R_k into R_0
`jump` n	$PC := n$	Set the program counter to n
`jzero` n	if $R_0 = 0$ then $PC := n$	Set the program counter to n, if R_0 is zero
`jpos` n	if $R_0 > 0$ then $PC := n$	Set the program counter to n, if R_0 is positive
`accept`		Accept
`reject`		Reject

Table 1: The instruction set of a Random Access Machine.

Most of these instructions are quite standard. Variants of them exist in virtually every computer architecture. The only exception are the two special instructions, which we call `accept` and `reject`, which are not useful for real computers. The reason we added them is because we want to compare our random access machine to the Turing Machine. Since the goal of the Turing Machine is to accept or reject its input, in order to compare the two models properly, we have to find a way to say what it means for a Random Access Machine to accept or reject its input. We will do this by executing these special commands.

The other issue we need to settle on before we can compare Turing Machines and Random Access Machines is how we are going to give the machine an input to work on. In practice, computers get their input from external devices, like keyboards, touchscreens, scanners, the internet, and so on. However we do not have any provision for attaching input devices to the Random Access Machine. What happens on a computer when, say, you type something on your keyboard? Usually, computers goes into a special mode where the CPU execution is interrupted and the keyboard input is copied into memory before the CPU resumes its execution. So for our random access machine, it is reasonable to assume that before we start execution, the input will already be present in memory. For instance, if the input is k symbols long, we will assume it is present in the first k memory locations $M(0), M(1), \ldots, M(k-1)$.

One more technicality concerns the alphabet. The Turing Machine works with an arbitrary alphabet Γ, while the Random Access Machine has no such alphabet; it can only work with integers. However we can represent symbols in Γ easily using integers in the set $\{0, \ldots, |\Gamma| - 1\}$. By convention, we will use the integer 0 to represent the blank tape symbol of the Turing Machine.

To summarize the previous points, here is how an execution of the Random Access Machine proceeds:

1. Initially, the input $x \in \Sigma^*$ is loaded in the first k cells of the memory $M(0), \ldots, M(k-1)$, where k is the length of x. The registers $R_1, \ldots, R_m$, program counter PC, and the other

memory cells $M(k), M(k+1), \ldots$ are initialized to the value 0.

2. At each step of the execution, the Random Access Machine executes the program line pointed to by the program counter PC. After executing the instruction, the value of PC is incremented by 1, unless the instruction is `jump`, `jzero`, `jpos`, `accept`, or `reject`. If the instruction is `jzero` or `jpos` and the condition is not satisfied, PC is also incremented by 1.

3. If the current instruction is `accept`, the Random Access Machine accepts and halts. If the current instruction is `reject`, the Random Access Machine rejects and halts.

2 Simulating a Turing Machine on a Random Access Machine

Now we will argue that the Random Access Machine can do at least what a Turing Machine can do. This should not be very surprising, because the Turing Machine is a very primitive computer. Surely we expect the Random Access Machine, which represents a more realistic computer, to be able to do whatever a Turing Machine can do.

To explain why this is the case, we will outline how to do a simulation of any Turing Machine on some Random Access Machine. To do this, we have to give a way to represent the *configuration* of the Turing Machine (its internal state, its tape contents, and the position of its head) on a Random Access Machine. Then we have to show how to simulate each step of the Turing Machine on the Random Access Machine in a way that preserves the representation of this configuration.

There are various ways to represent the configuration of a Turing Machine on a RAM. We can do as follows. The tape of the Turing Machine will be represented in the memory of the RAM, so that memory cell $M(i)$ contains the contents of the ith location of the Turing Machine tape. The state of the Turing Machine will be represented in the program counter PC of the RAM, while the head location will be stored in the register R_0.

Now we have to say what the RAM should do when the Turing Machine performs a transition. To do this, we need to write a set of instructions for the RAM which will simulate the corresponding transition in the Turing Machine.

Specifically, suppose we have arranged our simulation so that when the Turing Machine is in state q_i, the value of the PC is p_i. Then the instructions p_i, p_i+1, p_i+2, up to $p_{i+1}-1$ will be used to simulate transitions out of the state q_i. For the initial state q_0, we start at line $p_0 = 0$.

First, we want to look at the current tape symbol. Since the head position is stored in R_0, we can do this by executing the command `read R0`. However, we have to be careful because this will overwrite the contents of R_0. To be safe we copy these into R_1 first:

line	**instruction**	**meaning**
p_i:	`store R1`	Save head position into R_1
p_i+1:	`read R1`	Put the current tape symbol into R_0

Now we look at the transition of the Turing Machine. Suppose we have a transition that says

$$\delta(p_i, 1) = (p_j, 2, R)$$

meaning, if we see 1 on the tape in the current state p_i, we replace it with a 2, move the tape head to the right, and move to state p_j. We can implement that transition by the following program fragment:

$p_i + 2$:	`add -1`	If the tape symbol was a 1, R_0 becomes 0
$p_i + 3$:	`jzero` $p_i + 10$	If this is the case, go to line $p_i + 10$

At line $p_i + 10$, we write the code for this transition: Replace the 1 on the tape with a 2 and move the head right.

$p_i + 10$:	`load 2`	New value to be stored on tape
$p_i + 11$:	`write R1`	Write the new tape symbol
$p_i + 12$:	`load R1`	Move back the tape head into R_0
$p_i + 13$:	`add 1`	Move the tape head to the right
$p_i + 14$:	`jump` p_j	Go to state p_j

To handle the next transition, which maybe says $\delta(p_i, 2) = (p_l, 2, L)$, we write the next program snippet starting at line p_{i+4}:

$p_i + 4$:	`add -1`	If the tape symbol was a 2, R_0 now becomes 0
$p_i + 3$:	`jzero` $p_i + 10$	If this is the case, go to line $p_i + 15$
	...	
$p_i + 15$:	`load 2`	New value to be stored on tape
$p_i + 16$:	...	

and so on. After we have finished all of these we have a complete program for the transitions of the Turing Machine. If state q_i is the accept state, we only write the line

p_i:	`accept`	Turing Machine has accepted

and similarly for the reject state.

3 Simulating a Random Access Machine on a Turing Machine

To simulate a RAM on a Turing Machine, we have to decide how we are going to represent the state of the RAM (its registers and memory) on the Turing Machine tape. For convenience, we can use a Turing Machine with multiple tapes. We represent each of the m registers on tapes 1 up to m. Tape $m + 1$ will represent the memory of the RAM. Finally, we will have another tape $m + 2$ which we will use for scratch work in order to update the first $m + 1$ tapes when necessary.

One way to represent the memory is as a list of the form $(a_1, v_1)(a_2, v_2) \dots (a_t, v_t)$, where (a_i, v_i) means that the memory cell a_i has value v_i. Since the memory is infinite, but we can only represent a finite number of cells, those whose value is zero will be omitted from the list. For example, if the memory contents are

memory:

3	-1	0	-1	0	0	···

then the memory will be represented by the string $(0, 3)(1, -1)(3, -1)$ on the memory tape (tape $m + 1$) of the Turing Machine.

Now we have to explain how the simulation proceeds. Initially, the Turing Machine starts with the input on its first tape. In order to simulate a RAM machine, we need to convert the input as if it were written on the RAM – namely, rewrite it on the memory tape in proper form. Moreover, we have to ensure the first m tapes, representing the registers, start with value zero. Here is a high-level description of this initialization step of the Turing Machine:

Initialization: On input $x_1 \dots x_n$:

1. Insert a special marker `$` at the beginning of each tape.
2. Write the string $(\texttt{0}, x_1)(\texttt{1}, x_2) \cdots (n-1, x_n)$ on the memory tape (tape $m+1$) of the Turing Machine. Use the scratch tape to keep track of the memory addresses $0, 1, \dots, n-1$ as you go along.
3. Erase the contents of the PC tape (tape 1) and write the value `0` on it. Write values `0` on all the register tapes (tapes 1 up to m).
4. Erase the contents of the scratch tape (except for the special marker).
5. Move to the state representing instruction 0 of the RAM program.

Now that the input is converted to the proper form, we can simulate the instructions of the RAM on the Turing Machine. For each instruction of the RAM, there will be a set of states of the Turing Machine whose task it is to simulate that instruction. For each such set of states, there will be an initial state which "represents" the corresponding instruction. We now give some examples of how this simulation is done.

Suppose we need to simulate the instruction `load` n, which is part of the program of the RAM. Recall that this instruction is supposed to put the value n in register R_0. To do so, the look at the R_0 tape (tape 1) of the Turing Machine, erase its current value, and write the value n on it. We then move to the state of the Turing Machine that describes the next instruction in the program.

Now suppose instead we need to simulate the instruction `loadR`k, which is supposed to copy the value of R_k into R_0. To do this on the Turing Machine, we erase the R_0 tape, and copy the contents of the R_k tape onto the R_0 tape. We then move to the state of the Turing Machine that describes the next instruction in the program. If instead we want to `storeR`k, we do the same operation with the roles of the R_0 and R_k tapes reversed.

Now let us consider the instruction `readR`k. This instruction is supposed to copy the value at memory location R_k into R_0. To simulate it, first we erase the R_0 tape. We then walk along the memory tape and compare each address component a_i with the contents of the R_k tape. If we find a match, we then copy the corresponding part v_i of the $(a_i, v_i$ pair onto the R_0 tape. If we do not find R_k among the addresses a_i, we write 0 on the R_0 tape. We then move to the state of the Turing Machine that describes the next instruction in the program.

For the operation `writeR`k, we first attempt to find the address value described by tape R_k on the memory tape. If we found such a value and the R_0 tape has value 0, we erase the pair (a_i, v_i) from the memory tape (and shift the subsequent contents to the left so it is properly formatted). If we

found a value a_i that matches R_k and the R_0 tape has a nonzero value, then we replace the value v_i with the value on the R_0 tape. If we did not find an address value for R_i, then we make a new pair (a_i, v_i) on the memory tape, where we copy a_i from the R_k tape and we copy v_i from the R_0 tape. Finally, we move onto the state representing the next instruction.

For the instruction `add` n, `addR`k, and `multR`k, we need to perform some arithmetic. To do so, we copy the values of R_0 and R_k (or n) on the scratch tape. We then perform the addition or multiplication on the scratch tape. We then copy the contents back to the R_0 tape, and move to the state representing the next instruction.

For the instruction `jump` n, we simply move to the state of the Turing Machine that represents instruction n. For `jzero` n (resp., `jpos` n), we first look at the R_0 tape, check the condition, and then jump to the state representing the correct next instruction.

Finally, the instructions `accept` and `reject` are represented by the accept and reject states of the Turing Machine, respectively.