§5. Random Access Machines (RAMs). Over the years, many different formulations of the notion of effective computability have been proposed. These formulations differ widely in appearance; nevertheless, whenever a new formulation has been proposed, it has always turned out to be equivalent to all previous ones. Our failure to find a reasonable model of computation which properly extends the notion of effective computability provides what is perhaps the most convincing empirical evidence in support of Turing's thesis.

In this note, we consider a model of computation very different from the Turing machine, namely the *Random Access Machine*, or *RAM*. In the context of the *Computability and Intractability* module, the significance of the RAM is that it provides a relatively simple basis for computation which is not far removed from real computers. Thus, in introducing the RAM, we bring into play all the intuition about computation we have gained in the past two (or more) years.

As its name suggests, the main feature of the RAM is its ability to access data *by address*, rather than merely *sequentially* as in a Turing machine. (The relationship between the two models can be likened to the relationship between *array* and *doubly-linked list* as data-structuring techniques.) Despite the apparently less restricted nature of computation on a RAM, we shall show, in due course, that the RAM and Turing machine are of equivalent power.

§5.1. Syntax of RAM programs. The syntax of a RAM program is presented below, in Backus-Naur form.

$$
\begin{aligned}
\mathit{program} &= \mathit{instruction}\ \mathit{program} \mid \mathit{instruction} \\
\mathit{instruction} &= [\,\mathit{label}\,\texttt{:}\,] \,(\texttt{accept} \\
&\qquad\qquad \mid \texttt{reject} \\
&\qquad\qquad \mid \texttt{read}\ \mathit{l\_value} \\
&\qquad\qquad \mid \mathit{l\_value}\ \texttt{:=}\ \mathit{r\_value}\ \mathit{arithmetic\_op}\ \mathit{r\_value} \\
&\qquad\qquad \mid \texttt{if}\ \mathit{r\_value}\ \mathit{relational\_op}\ \mathit{r\_value}\ \texttt{goto}\ \mathit{label}\,) \\
\mathit{l\_value} &= \texttt{'}\mathit{integer} \mid \texttt{"}\mathit{integer} \\
\mathit{r\_value} &= \mathit{integer} \mid \texttt{'}\mathit{integer} \mid \texttt{"}\mathit{integer} \\
\mathit{arithmetic\_op} &= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{div} \\
\mathit{relational\_op} &= \texttt{=} \mid \texttt{<>} \mid \texttt{<=} \mid \texttt{<} \\
\mathit{label} &= \mathit{alphanumeric\_sequence}.
\end{aligned}
$$

The non-terminal *integer* is intended to stand for an arbitrary signed decimal (whole) number. Naturally enough, we insist that no two instructions are assigned the same label, and that every conditional jump refers to an existent label.

We shall assign a precise meaning to RAM programs in the following section. Roughly speaking, however, an unquoted integer denotes a *constant*, an integer prefixed by a single quote is to be interpreted as a *direct address*, and an integer prefixed by a double quote is

to be interpreted as an *indirect address*. Armed with this clue, the reader should already be able to guess the meaning of many, if not all the instruction types.

§5.2. Semantics of RAM programs. We assign meaning to a RAM program by introducing the notion of state of a random access machine. Informally, a RAM has an infinite number of registers, each containing an arbitrary integer. The registers are indexed by the integers; the index of a register is sometimes called its *address*. Before execution of a program, all registers contain zero. The instructions of a program are thought of as being numbered in sequence, starting at zero. A particular instruction of the program is picked out by an *instruction counter*, which is a natural number; this instruction counter is initialized to zero. Execution of the program proceeds in a sequence of steps. In a single step, the instruction indicated by the instruction counter is executed, as a result of which the state of the machine and the instruction counter are updated. The state and instruction counter together play the role of configuration in the Turing machine model.

Formally, the *state* of a RAM is a function $s : \mathbb{Z} \to \mathbb{Z}$, which defines the content $s(i)$ of each register i. The function s has a finite description, being zero on all but a finite subset of $\mathbb{Z}$. The initial state of a RAM (before the program executes) is the zero function. Before describing the effect of executing each instruction type, it is convenient to introduce some terminology and notation. If R is an *r_value* and s a state, then the *result of evaluating* R *in context* s is an integer value v given by

$$v = \begin{cases} k, & \text{if } R = k; \\ s(k), & \text{if } R = \texttt{'}k; \\ s(s(k)), & \text{if } R = \texttt{"}\,k. \end{cases}$$

If L is an *l_value* and s a state, then the *result of evaluating* L *in context* s is an integer address a given by

$$a = \begin{cases} k, & \text{if } L = \texttt{'}k; \\ s(k), & \text{if } L = \texttt{"}\,k. \end{cases}$$

Finally, if s is a state, v an integer value, and a an integer address, then update(s, v, a) denotes the new state $s' : \mathbb{Z} \to \mathbb{Z}$ given by

$$s'(i) = \begin{cases} v, & \text{if } i = a; \\ s(i), & \text{otherwise.} \end{cases}$$

Informally, the new state s' agrees with the old state s at all points except a, where $s'(a) = v$ independent of the value of $s(a)$.

We are now in a position to assign a meaning to each instruction by specifying how that instruction modifies the state and instruction counter. Let s denote the state before execution of the instruction in question, and s' denote the state afterwards.

(a) `accept`: The RAM halts, and is deemed to have accepted its input.

(b) `reject`: The RAM halts, and is deemed to have rejected its input.

(c) `read` L: The input to a RAM is a stream of integers. The next value, say v, is removed from the stream. Let a be the result of evaluating L in context s. Then s' becomes update(s, v, a), and the instruction counter is incremented by one.

(d) L `:=` $R_1 \circ R_2$: Let a, v_1, and v_2 be the results of evaluating L, R_1, and R_2 in context s, and let $v = v_1 \circ v_2$.[13] Then s' becomes update(s, v, a), and the instruction counter is incremented by one.

(e) `if` $R_1 \circ R_2$ `goto` λ: Let v_1 and v_2 be the results of evaluating R_1 and R_2 in context s. If $v_1 \circ v_2$ is false, the instruction counter is incremented by one.[14] If $v_1 \circ v_2$ is true, the instruction counter is set to the index of the instruction labelled by λ. In either case the state is unchanged, i.e., $s' = s$.

After executing the (syntactically) last instruction of a program, the instruction counter may no longer contain a meaningful value; in that case the RAM halts and rejects.

A random access machine M of the form described above can be viewed as a language acceptor. Let Σ be a finite input alphabet, and associate the symbols of Σ with the numbers $1, 2, 3, \ldots, |\Sigma|$. Then a word $x \in \Sigma^n$ can be presented to the RAM as a sequence of n positive numbers (encoding elements of Σ) followed by 0 (which can be thought of as an end-of-input marker or blank symbol). The *language $L(M)$ accepted by M* is then the set of words $x \in \Sigma^*$ on which M halts and accepts. It is a straightforward task to extend the model to encompass *transducers* by adding a instruction of the form '`write` *r_value*' to the repertoire of instructions.

§5.3. Example: recognizing palindromes. The RAM program in the accompanying figure accepts the language of palindromes over $\{a, b\}$, where a is encoded as 1, and b as 2. The n symbols of the input are read into registers 2 to $n + 1$, which can be thought of as constituting an n-element array. Registers 0 and 1 are used to implement indices, i and j say, into this array. Initially, $i = 2$ and $j = n + 1$. At each iteration, the array elements indexed by i and j are compared. If these elements are found to be unequal then the input was *not* a palindrome and the program halts and rejects. Otherwise the index i is incremented, and j decremented. If the pointers cross (i.e, j becomes less than or equal to i) then the input *was* a palindrome and the program halts and accepts.

§5.4. Redundancy. The RAM model described in this note contains a fair number of redundant features. It is not too difficult to show that the arithmetic operators `+`, `*`, and `div` can be removed without affecting the class of languages which can be recognized. Likewise, the relational operators `=`, `<>`, and `<` are redundant. More surprisingly, as we shall see later, it is possible to make do with a fixed, finite set of registers, and dispense with indirect addressing entirely!

[13] The operator `*` denotes integer multiplication and `div` denotes integer division. Thus `div` takes two integers v_1 and v_2, with $v_2 > 0$, and yields the unique integer v satisfying $0 \leq v_1 - vv_2 < v_2$; if $v_2 \leq 0$ the program halts and rejects.

[14] The relational operator `<>` means 'not equal' while `=`, `<=`, and `<` have the obvious meaning.

```
                '1 := 2
next_symbol:    read "1
                if "1 = 0 goto end_of_input
                '1 := '1 + 1
                if 0 = 0 goto next_symbol
end_of_input:   '1 := '1 - 1
                '0 := 2
loop:           if '1 <= '0 goto yes
                if "0 <> "1 goto no
                '0 := '0 + 1
                '1 := '1 - 1
                if 0 = 0 goto loop
yes:            accept
no:             reject
```

Figure 5: A RAM program for recognizing palindromes

§5.5. Bounded RAMs. A *RAM with bounded registers* differs from the conventional RAM defined above in only one respect: rather than containing arbitrary integers, the registers are now restricted to contain integers in some bounded range $\{-N, -N+1, \ldots, N-1, N\}$ (where N is fixed for any given machine but can vary from one machine to the next). More formally, the *state* of a RAM with bounded registers is a function $s : \mathbb{Z} \to \{-N, -N+1, \ldots, N-1, N\}$. Note that the *number* of registers is still infinite. Clearly bounded RAMs are no more powerful than ordinary ones; anything accepted by a bounded RAM can be accepted by an unbouded one. However the converse relation is not immediately clear. It is possible to provide an informal argument that seems to show that bounded RAMs are as powerful as ordinary ones; however this is not the case and we proceed to prove this.

Let P be a program with l instructions for a RAM with bounded registers. At any point in the execution of P, how many registers can contain a value other than zero (the initial value)? Well, at most l registers may be directly addressed via instructions of the form `read '`k and `'`k := ?, and a further $2N+1$ registers may be accessed indirectly via instructions of the form `read "`k and `"`k := ?, making a total of $2N+l+1$ in all. Let a 'configuration' of the RAM be the combination of the *state* and *instruction counter*. The total number of configurations is bounded by $M = l(2N+1)^{2N+l+1}$.

A 'pumping lemma' style argument now establishes that no such program P can recognize, for example, the language of all binary palindromes. Suppose, to the contrary, that there did exist such a P. Let n satisfy $2^n > M$, and consider the computation of P on inputs of the form xx^R where x is a binary word of length n (x^R denotes the string x written in reverse). In particular, consider the configuration of the RAM at an instant just before the $(n+1)$th input symbol is read. By the pigeonhole principle there must be distinct words x and y such that the inputs xx^R and yy^R drive P into the same configuration. But

then P would accept input xy^R which is not a palindrome.

The argument given above makes use of the following simple observation: suppose we have a finite set $\mathcal{C}$ (in our case this consists of configurations) and a (partial) function $d : \mathcal{C} \to \mathcal{C}$ (in our case this is the next move, if any, from a given configuration). Suppose $d(C), d(d(C)), \ldots$ are all defined for $C \in \mathcal{C}$ and consider a sequence:

$$\begin{aligned} C_1 &= C, \\ C_2 &= d(C_1), \\ C_3 &= d(C_2), \\ &\vdots \\ C_i &= d(C_{i-1}), \\ &\vdots \end{aligned}$$

If the sequence is continued for long enough (so that $d(C_i)$ is defined each time) then it is bound to repeat, i.e., there will be indices $1 \leq r < s$ such that $C_r = C_s$.

EXERCISE What is the longest, in terms of $|\mathcal{C}|$, that we have to wait before a repeat in the sequence given above?

§5.6. Equivalence of Turing machines and RAMs. None of us has yet conceived of a procedure which could reasonably be described as effective, but which could not be expressed in a high-level programming language such as C. Our failure to find such an object—in over two years of programming—can be regarded as empirical evidence in favour of the proposition that every effective procedure could, in principle, be expressed in C.

Next, we are aware, from the CS2 course, that any C program can be translated into machine code, as long as we set aside the limitations inherent in a *bounded* word-size and *bounded* address-space. This, of course, is exactly the job of a C compiler, and we have definite evidence that such things exist. Finally, the RAM model has at least the power of a conventional machine code, but without the restrictions which attach to bounded word-size or address-space. We thus have convincing empirical evidence that any effective procedure can be expressed as a RAM program.

Our experience with Turing machines may be closer to a few weeks than two years, and we may be less convinced that every effective procedure may be described by a Turing machine. Surely any such doubts will be dispelled if we demonstrate that Turing machines are at least as powerful as RAMs, thus providing convincing empirical evidence in support of Turing's Thesis. (Refer to NOTE 2.) The claim is made precise in the following theorem.

THEOREM 5.1 *Let L be a language over some alphabet Σ. If there is a RAM that accepts L, then there is a Turing machine that also accepts L.*

In fact Turing machines are no more powerful than RAMs as can shown by proving the converse to the preceding theorem. It is interesting that rather more can be demonstrated.

A *three-register RAM* is a random access machine, as defined above, but having just three registers, with addresses -1, 0, and 1. The *state* of a three register RAM is thus a function from $\{-1, 0, 1\}$ to $\mathbb{Z}$. To avoid referencing non-existent registers we place a severe restriction on the form of *l_values* and *r_values* that may occur in a program for a three-register RAM: the only *l_values* allowed are `'-1`, `'0`, and `'1`; and the only *r_values* allowed are `'-1`, `'0`, `'1`, and signed decimal constants. Note that 'indirect addressing' is forbidden.

THEOREM 5.2 *Let L be a language over some alphabet Σ. If there is a Turing machine that accepts L, then there is a three-register RAM that also accepts L.*

The proof of Theorem 5.1 is given in APPENDIX B and that of Theorem 5.2 in APPENDIX C. (We note that both theorems hold if we regard the machines as transducers rather than language acceptors.) The proofs are constructive in that they show us how, given a machine of one type that accepts L, we can produce a machine of the other type that also accepts L (cf. the proof of Theorem 4.1).

§5.7. Recursively enumerable languages. It is a good time to take stock of the situation. Let $\mathcal{C}_{\text{TM}}$ (respectively, $\mathcal{C}_{\text{RAM}}$, $\mathcal{C}_{\text{3RAM}}$) be the class of all languages that are accepted by some Turing machine (respectively, RAM, three-register RAM). We know from Theorem 5.1 that $\mathcal{C}_{\text{RAM}} \subseteq \mathcal{C}_{\text{TM}}$, and from Theorem 5.2 that $\mathcal{C}_{\text{TM}} \subseteq \mathcal{C}_{\text{3RAM}}$. Furthermore, since a three-register RAM is a special case of a RAM, it is clear that $\mathcal{C}_{\text{3RAM}} \subseteq \mathcal{C}_{\text{RAM}}$. It follows from these three inclusions that $\mathcal{C}_{\text{TM}} = \mathcal{C}_{\text{RAM}} = \mathcal{C}_{\text{3RAM}}$. Thus, three language classes defined in very different ways turn out to be the *same* class. This observation strongly suggests that the class of languages has a special significance, and warrants a special name. For historical reasons (which we shall not pursue here) languages in $\mathcal{C}_{\text{TM}}$ are called *recursively enumerable* (usually abbreviated to r.e.), and $\mathcal{C}_{\text{TM}}$ is the *class of recursively enumerable languages*. Note that the equality of the three language classes $\mathcal{C}_{\text{TM}}$, $\mathcal{C}_{\text{RAM}}$, and $\mathcal{C}_{\text{3RAM}}$ is further compelling evidence in support of Turing's thesis (NOTE 2).