

Sender–Receiver Exercise 3: Reading for Senders

Harvard SEAS - Spring 2026

2026-02-23

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to practice reasoning about the equivalence of more involved variants of the Word-RAM model via simulations

1 Recap: Word-RAM model

Please review the definition of Word-RAM model, part of which was covered in class.

Definition 1.1. The *Word-RAM program* P is defined like a RAM program except that it has a (static) word length parameter w and a (dynamic) *memory size* S .

- The word length w is *static*, meaning that while it can be set to an arbitrary value in \mathbb{N} prior to running P on an input x , it does not change during the computation.
- The memory size S is *dynamic*, meaning that it can change during the computation, as described below.

These are used as follows:

- **Memory:** the memory is an array of length S , with entries in $\{0, 1, \dots, 2^w - 1\}$. Reads from and writes to memory locations larger than S have no effect. Initially $S = n$, the length of the input array x . Additional memory can be allocated via a `MALLOC` command, which increments S by 1.

- **Output:** if the program halts, the output is defined to be

$$(M[\text{output_ptr}], M[\text{output_ptr} + 1], \dots, M[\min\{\text{output_ptr} + \text{output_len} - 1, S - 1\}]).$$

That is, portions of the output outside allocated memory are ignored.

- **Operations:** Addition and multiplication are redefined from RAM Model to return $2^w - 1$ (the max possible value) if the result would be $\geq 2^w$.
- **Crashing:** A Word-RAM program P *crashes* on input x and word length w if any of the following occur:
 1. One of the constants c in the assignment commands (`var = c`) in P is $\geq 2^w$.
 2. $x[i] \geq 2^w$ for some $i \in [n]$
 3. $S > 2^w$. (This can happen because $n > 2^w$, or if $2^w - n + 1$ `MALLOC` commands are executed.)

We denote the computation of a Word-RAM program on input x with word length w by $P[w](x)$. Note that $P[w](x)$ has one of three outcomes:

- halting with an output
- failing to halt, or
- crashing.

We define the *runtime* $T_{P[w]}(x)$ to be the number of commands executed until P either halts or crashes (so $T_{P[w]}(x) = \infty$ if $P[w](x)$ fails to halt).

1.1 A 2D WordRAM model

We define a 2-D Word-RAM model, as similar to the Word-RAM model, but the memory is a two-dimensional square array with side length S (hence total size $S \times S$), instead of a one-dimensional array. The changes in the definition are listed below; all other specifications stay the same.

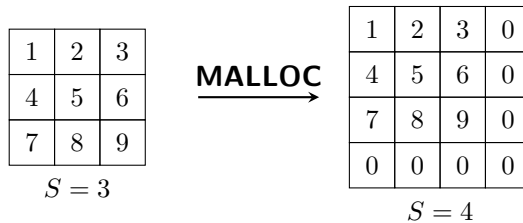
- The commands to write to and read from memory use two variables for memory addresses:
 1. (read from memory) $\text{var}_0 = M[\text{var}_1][\text{var}_2]$ for variables $\text{var}_0, \text{var}_1, \text{var}_2 \in V$.
 2. (write to memory) $M[\text{var}_0][\text{var}_1] = \text{var}_2$ for variables $\text{var}_0, \text{var}_1, \text{var}_2 \in V$.

As before, reads and writes to memory locations outside the $S \times S$ array have no effect.

- The parameter S is now called *memory width*. The crash conditions stay the same as in usual Word-RAM, including a crash if $S > 2^w$.
- Input is constrained to be within the first row and the variable `input_len` specifies the length of the input. We set $S = \text{input_len}$ in the initialization step, which suffices to include the input in the available memory.
- Output is defined to be in the first row, in the cells

$$M[0][\text{output_ptr}], \dots, M[0][\min\{\text{output_ptr} + \text{output_len} - 1, S - 1\}].$$

- MALLOC increases the memory width S by 1 (and consequently increases memory size from S^2 to $(S + 1)^2 = S^2 + 2S + 1$).



The goal of this exercise is to understand the proof of the following theorem.

Theorem 1.2. *For every 2-D Word-RAM program P , there is a 1-D Word-RAM program P' such that the following holds.*

1. For every input x and every word length w such that $P[w](x)$ does not crash, there is a word length w' such that $P'[w'](2^w, \max_i x[i], x)$ halts without crashing on x iff $P[w](x)$ halts, and if they do halt,

$$P'[w'](2^w, \max_i x[i], x) = P[w](x).$$

2. If $P[w](x)$ crashes for some x, w , then $P'[w'](2^w, \max_i x[i], x)$ crashes for all possible w' .
3. If $P[w](x)$ halts without crashing in t steps, then there is a w' such that $P'[w'](2^w, \max_i x[i], x)$ halts without crashing in $O((n + t)^2)$ steps. Here n is the size of the input x .

1.2 The Proof

We provide an explicit construction of the Word-RAM program P' . Let the memory of P' be M' (to distinguish it from the memory of P , which is denoted M). Note that P' is working on inputs $2^w, \max_i x[i], x$, which is provided in the memory locations $M'[0], \dots, M'[\text{input_len}' - 1]$. We denote the memory size of P' as S' , to distinguish it from the memory width S of P . We will view the memory location $M[\text{var}_1][\text{var}_2]$ in P with the memory location $M'[S' \cdot \text{var}_1 + \text{var}_2 + 2]$ in P' (the $+2$ comes from having to store $2^w, \max_i x[i]$, that requires two extra cells). As in the statement of the theorem, the word length of P' is w' .

1. **Crashing sweep:** The initial value of S' is set by default to `input_len`. We perform basic tests to see if the choice of $2^w, \max_i x[i]$ would already cause a crash. We check if $2^w \leq \text{input_len}' - 2$ (since the length of x is `input_len' - 2`), or $\max_i x[i] \geq 2^w$, or if any constants used on variable assignments are larger than $2^w - 1$. If so, we crash the program by repeatedly calling `MALLOC`.
2. **Initialization:** We define a variable `2D_memwidth = input_len - 2` (equal to the size of the input x). Call `MALLOC (2D_memwidth)2 + 2 - input_len` times to ensure that the memory size for the Word-RAM program is $S' = (2D_memwidth)^2 + 2$. Note that this step may cause a crash if w' were not large enough (which is okay! - see correctness proof below).
3. **Read from Memory:** The command `var0 = M[var1][var2]` is replaced as follows. If either of `var1, var2` are larger than `2D_memwidth`, we move to the next line of code (as we need to ignore any reads and writes to memory that are out of the scope) . Otherwise, we proceed as follows:

```
var3 = 2D_memwidth × var1
var4 = var3 + var2 + 2
var0 = M'[var4]
```

4. **Write to Memory:** `M[var0][var1] = var2` is replaced as follows. If `var1, var2` are larger than `2D_memwidth`, we move to the next line of code (as we need to ignore any reads and writes to memory that are out of scope) . Otherwise, we proceed as follows:

```
var3 = 2D_memwidth × var0
var4 = var3 + var1 + 2
M[var4] = var2
```

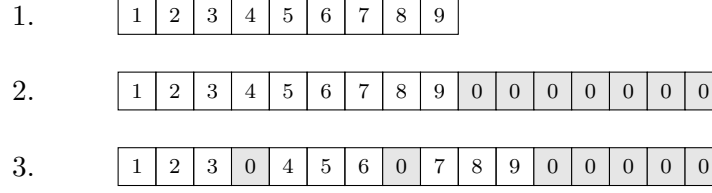


Figure 1: A depiction of the pseudocode for MALLOC. You are encouraged to work out with your partner on what the `2D_memwidth` is at pictures 1, 2, and how do the lines 1-3 of the pseudocode perform the move. Keep in mind that the cell with entry 1 starts at memory location 2, as memory locations 0, 1 contain the extra data $2^w, \max_i x[i]$ not originally present in input to P .

5. **MALLOC:** Every MALLOC command in P is replaced by the following.

We increase `2D_memwidth` by 1. If `2D_memwidth` $\geq 2^w$, we keep calling MALLOC (which will cause crash at some point). Else, call MALLOC $2 \cdot \text{2D_memwidth} + 1$ times (to maintain $S' = (\text{2D_memwidth})^2 + 2$). Now, we need to move the values in the memory to maintain the association between the 2D memory M and the one dimensional memory M' . This is done by the Word-RAM equivalent of the following pseudocode ¹.

```

1 foreach  $var_1 = 2D\_memwidth - 2, \dots, 0$  do
2   foreach  $var_2 = 2D\_memwidth - 2, \dots, 0$  do
3     temp =  $M'[(2D\_memwidth - 1) * var_1 + var_2 + 2]$ ;
       $M'[(2D\_memwidth) * var_1 + var_2 + 2] = temp$ .
```

The graphical description of this pseudocode is in Figure 1

6. **Operations:** For addition and multiplication operations, we need to truncate the outputs appropriately if needed. We use the input 2^w to additionally check if the result of an addition or multiplication would exceed $2^w - 1$. If so, we set it to $2^w - 1$.

7. **Conditional GOTO:** we update all the line numbers in the conditional GOTO statements in P' to make sure that they go to the equivalent command as in P .

This completes the specification of the Word-RAM program P' .

Now, we discuss the proof of correctness and runtime.

Correctness: We establish Parts 1,2 in the statement of Theorem 1.2. Part 2 holds simply because we have kept track of crashing conditions on P in the construction of P' above, and induced a crash whenever needed by calling MALLOC enough times. So, let's consider Part 1. Suppose P does not crash on x for some word-length w . Operations in Items 2-7 above do not lead to crash for large enough w' . The output $P'[w'](x, 2^w, \max_i x[i])$ is the same as $P[w](x)$, since each item simulates the corresponding operation in P for large enough w' (we are performing various arithmetic computations in each item listed above; one can verify by inspection that they will not be truncated to $2^{w'} - 1$ for large enough w').

¹we will skip the precise implementation of this pseudocode in Word-RAM for ease of understanding. You can convince yourself that this is possible by appealing to the fact that the pseudocode can be implemented in Python, which can in-turn be implemented in Word-RAM model.

Runtime: Here, we show Part 3 of Theorem 1.2. The items 2-4,6-7 only increase $O(1)$ lines of code for each line of code in P . Item 1 will cause additional runtime of $O(2^{w'})$ as enough `MALLOC` will be needed to cause crash. Item 5 is the most expensive step and the number of times the loop runs is $(2\text{D_memwidth})^2$. Within each loop, we have $O(1)$ operations, which gives the effective runtime of $O(2\text{D_memwidth})^2$. We can upper bound this number by $O((n+t)^2)$ as `2D_memwidth` will not exceed $n+t$.