1 RAM

A word-RAM consists of:

- A fixed set of instructions $P_1, \ldots, P_q$. Allowed instructions are:
  - Modular arithmetic and integer division on registers; the standard model for a RAM machine does not have negative numbers, but you can verify that allowing signed integers doesn't chance the power of the model.
  - Bitwise operations on registers.
  - Loading a constant into a register.
  - Transfering values between a register and a word in main memory (as addressed by another register).
  - Conditional GOTO.
  - MALLOC – increase the size S of main memory by 1. This also increments the word size if the current word size is too small to address all of main memory.
  - HALT.
- The program counter $l \in \{1, \ldots, q\}$, telling us where we are in the program. Except for GOTO statements, the counter increments after each instruction is executed.
- The space usage S, which starts at the size of the input.
- The word size w; the initial word size is $\lceil \log_2 \max(n+1, k+1) \rceil$, where n is the length of the input and k is the maximum constant appearing in the program or input.
- A constant number of registers $R[0], \ldots, R[r-1]$, each of which is a w-bit word.
- Main memory $M[0], \ldots, M[S-1]$, each of which is a w-bit word.

The tuple (l, S, w, R, M) gives the *configuration* of the word-RAM.

We say that a word-RAM *solves* a computational problem $f : \Sigma^* \to 2^{\mathbb{N}^*}$ if the machine which starts with input x halts with some output in $f(x)$ as the set of characters in $M[0], \ldots, M[R[0]-1]$.

As we saw with Turing Machines, there are many adjustments we can make to a model which make it easier to reason about without significantly changing its power. Here's one for word-RAMs:

Exercise. *We define a 2-D word-RAM model: Suppose that main memory, instead of being represented by a one-dimensional array of size S, is represented by a two-dimensional array of size $S \times S$. Consequently, saves or loads from main memory require addressing it using two registers; as before, MALLOC increases S by 1 (and consequently the size of memory by $2S+1$).*

Prove that a computational problem can be solved by a 2-D word-RAM model in polynomial time if and only if it can be solved by regular word-RAM model in polynomial time. In this problem, provide a ***formal description*** *of how to convert between the two models.*

Solution.
($\Longrightarrow$) **Intuition:** To simulate a 1-D word-RAM by a 2-D one, simply ignore all rows but the first, and whenever addressing some word in main memory, let the second index always equal 0. (That is, we keep one extra register in which we initially load the value 0, and then use that register only for the second index.)

Formal description: To turn this idea into a formal description, given a 1-D word-RAM program $P = (P_1, \ldots, P_q)$ with r registers, we have to give the code for an equivalent 2-D word-RAM program that works along the lines discussed above. Define the following equivalent 2-D word-RAM program $P' = (P'_0, P'_1, \ldots, P'_q)$: P has $r+1$ registers, and the instructions of P' are obtained largely by those of P in the following way (where we go over all possible instruction schemas, using the ordering in the lecture notes on word-RAMs):

1. Let P'_0 be $R[r] = 0$
2. For $i = 0, \ldots, q$:
   (a) If P_i is of the form $R[i] \leftarrow m$, let $P'_i = P_i$;
   (b) If P_i is of the form $R[i] \leftarrow R[j] + R[k]$, let $P'_i = P_i$;
   (c) ...
   (d) If P_i is of the form $R[i] \leftarrow \lfloor R[j]/R[k] \rfloor$, let $P'_i = P_i$.
   (e) If P_i is of the form $R[i] \leftarrow M[R[j]]$, let P'_i be $R[i] \leftarrow M[R[r]][R[j]]$
   (f) If P_i is of the form $M[R[j]] \leftarrow R[i]$, let P'_i be $M[R[r]][R[j]] \leftarrow R[i]$.
   (g) If P_i is of the form IF $R[i] = 0$, GOTO l, let P'_i be IF $R[i] = 0$, GOTO $l + 1$.
   (h) ...

So, apart from GOTO, which we have to change because we added a new row in the beginning, we only change the instructions having to do with memory access, by modifying them so that they act as if the first row of the 2-D word-RAM stands for the memory of the 1-D word-RAM. Notice that since MALLOC on the 2-D word-RAM restricts to the equivalent of MALLOC on the 1-D word-RAM on the first row under our identification, no change is required in the way the MALLOC operation works.

Running time and correctness: The fact that P' is correct follows by a straightforward induction on the number of instructions: after $t+1$ instructions of P' have been executed, when we restrict the memory of P' to the first row and the registers to the first r registers, we're in the same state as after t instructions of P have been executed.

For running time, notice that if P takes t steps to halt on some input, P' will take exactly $t + 1$ steps to halt. The wanted time bound follows directly.

($\Longleftarrow$) **Intuition:** To simulate a 2-D word-RAM by a 1-D one, we have to be careful about two things: simulating the MALLOC operation, and simulating the indexing into a two-dimensional array by indexing into a one-dimensional one.

For the MALLOC operation, we use an extra register to keep track of the value of S; then, when the 2-D word-RAM would do a malloc, we instead load $2S + 1$ into a second extra register (we may need a third one to hold the constants for this computation), and make that many MALLOCs in our 1-D simulation, decrementing the register as we go.

A convenient way to deal with the second problem, meanwhile, when the dimensions $S \times S$ are fixed, would be to let the entry $M_2[i][j]$ correspond to the entry $M_1[iS + j]$. However, S is not fixed in general, so if we want to keep with that nice formula we'd have to reorganize the memory contents every time we MALLOC.

To avoid this, we can index our 1-dimensional array in another way: let the cell [0][0] correspond to the cell 0, and then, think of expanding the square representing the 2-dimensional memory inductively, by increasing its current sidelength S by one, and appending the new cells $[S][0], [S][1], \ldots, [S][S], [S-1][S], \ldots, [0][S]$ to the 1-dimensional memory in this order. We can write down concrete formulas for this identification: the cell $[i][j]$ is represented by $[i^2 + j]$ if $i \geq j$ and $[j^2 + j + (j - i)]$ otherwise.

Formal description: We proceed in a way similar to what we did for easy direction; only now things require more careful adjusting.

So, to turn our idea into a formal description, given a 1-D word-RAM program $P = (P_1, \ldots, P_q)$ with r registers, we have to give the code for an equivalent 2-D word-RAM program P' that works along the lines discussed above.

One trickier thing we have to deal with, and one that our high-level discussion ignored, is GOTO statements: since we'll be substituting single instructions to the 2-D word-RAM with blocks of several instructions to the 1-D word-RAM, we have to be careful about the numbering of lines in our program.

So as we are converting the instructions (of which there are only constantly many, as we're working with a fixed program), we must keep track of how many of each kind we've seen so far, and with how many instructions we're replacing each, so that we know at which line we are, and to which line we want to get. Since we'll be replacing each instruction to the 2-D word-RAM of a given form with the same number of instructions to the 1-D word-RAM, this is a very doable task that we can do even before we've started conversion, just by counting the instructions of various types present in P. We'll assume we've done that, and suppress the explicit formulas for conversion in the code below, by letting $f(l)$ stand for the function giving the number of line l from the 2-D word-RAM program in the 1-D word-RAM program, and $\mathcal{L}$ stand for the current line number in the 1-D word-RAM program (you can think of these functions as of lookup tables).

We define the following 1-D word-RAM program P': P' has $r + 7$ registers, and the instructions of P' are obtained from those of P in the following way (where we go over all possible instruction schemas, using the ordering in the lecture notes on word-RAMs):

1. Let P'_0 be $R[r] \leftarrow R[1]$ (initialize current memory usage S, as $R[1]$ holds n)
2. Let P'_1 be $R[r + 5] \leftarrow 0$ (keep the constant 0 for ease of control flow)
3. Let P'_2 be $R[r + 6] \leftarrow 1$ (keep the constant 1)
4. For $i = 0, \ldots, q$: (replace instructions of P by suitable equivalents)
   (a) If P_i is of the form $R[i] \leftarrow m$, append P_i to the current program P'; (instructions that don't touch memory, or MALLOC, or GOTO, are left intact)
   (b) If P_i is of the form $R[i] \leftarrow R[j] + R[k]$, append P_i to the current program P'
   (c) ...
   (d) If P_i is of the form $R[i] \leftarrow \lfloor R[j]/R[k] \rfloor$, append P_i to the current program P'.

(e) If P_i is of the form $M[R[i]][R[j]] \leftarrow R[k]$ or $R[k] \leftarrow M[R[i]][R[j]]$, we append the following several instructions to the current program P':

i. $R[r+1] \leftarrow R[i]$ (prepare temporary variables)

ii. $R[r+2] \leftarrow R[j]$

iii. $R[r+3] \leftarrow \lfloor R[r+1]/R[r+2] \rfloor$ (test if $i < j$)

iv. IF $R[r+3] = 0$, GOTO $\mathcal{L}+4$ (in this case, $i < j$, so we skip ahead to the relevant computation)

v. $R[r+3] \leftarrow R[r+1] \cdot R[r+1]$ (this branch computes $R[i]^2 + R[j]$

vi. $R[r+3] \leftarrow R[r+3] + R[r+2]$

vii. IF $R[r+5] = 0$, GOTO $\mathcal{L}+5$ (this will always be true, as we keep $R[r+5] = 0$)

viii. $R[r+3] \leftarrow R[r+2] \cdot R[r+2]$ (this branch computes $R[j]^2 + R[j] + (R[j] - R[i])$)

ix. $R[r+3] \leftarrow R[r+3] + R[r+2]$

x. $R[r+3] \leftarrow R[r+3] + R[r+2]$

xi. $R[r+3] \leftarrow R[r+3] - R[r+1]$

xii. $M[R[r+3]] \leftarrow R[k]$, or $R[k] \leftarrow M[R[r+3]]$, according to which the form of the command we're replacing.

(f) If P_i is of the form IF $R[i] = 0$, GOTO l, append IF $R[i] = 0$, GOTO $f(l)$ to the current program P'.

(g) If P_i is of the form MALLOC, append the following several instructions to the program:

i. $R[r+4] \leftarrow R[r]$ (copy current memory usage S of the 2-D RAM, and increment it)

ii. $R[r] \leftarrow R[r] + R[r+6]$. (recall $R[r+6] = 1$)

iii. $R[r+4] \leftarrow R[r+4] + R[r+4]$ (compute $2S+1$)

iv. $R[r+4] \leftarrow R[r+4] + R[r+6]$

v. IF $R[r+4] = 0$, GOTO $\mathcal{L}+4$ (loop to do $2S+1$ MALLOC-s on 1-D word-RAM)

vi. MALLOC

vii. $R[r+4] \leftarrow R[r+4] - R[r+6]$

viii. IF $R[r+5] = 0$, GOTO $\mathcal{L}-3$

Running time and correctness: Correctness follows in a similar way as before; now the claim is that after t instructions in P, and the corresponding t' instructions in P', the two machines are in the same state under our identification between the 2-dimensional memory and the 1-dimensional memory via the formulas we wrote down in the beginning.

For running time, each instruction of P is simulated in $O(1)$ instructions of P', except for MALLOC, which takes $O(S)$ instructions, where S is the current sidelength of the square representing the memory of P. Since S can increase by at most 1 for each step of P, S is never more than the running time $T(n)$ of P, hence the running time of P' is never more than $O(T(n) \times O(T(n))) = O(T^2(n))$.

2 Turing Machines

Formally, a Turing machines is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{halt}})$, where

- Q is a *finite* set of states.
  - This means you can use your states to do *finite* counting (e.g. modulo 2), but not counting to arbitrarily large numbers.
  - Sometimes, instead of having a single q_{halt} state, people will talk about having a q_{reject} and q_{accept} state.
- Σ is the input alphabet.
- Γ is the tape alpabet, with $\Sigma \subset \Gamma$ and $\sqcup \in \Gamma - \Sigma$.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.
  - As we saw in class, we can simulate a multiple tape Turing machine with a single tape ; in this case, if we have k tapes, the transition function is $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.
  - We'll often also allow an S for staying still; this does not change the power of the TM.
- $q_0, q_{\text{accept}}, q_{\text{reject}} \in Q$ are the start state, accept state, and reject state. respectively

A Turing machine configuration is a string $uqv \in \Gamma^* Q \Gamma^*$ that encodes (i) the state q of M, (ii) the tape contents uv ignoring trailing blanks, and (iii) the location of the head within the tape. The starting configuration is $q_0 x$.

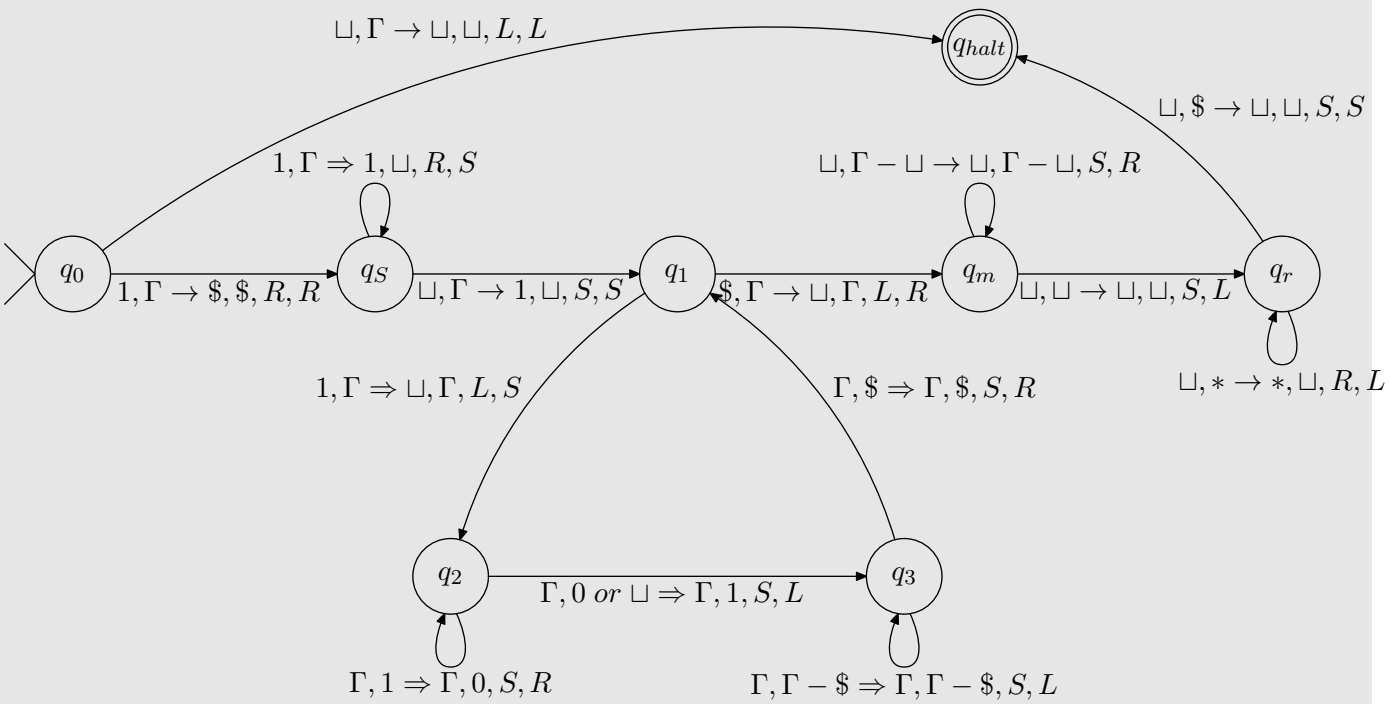
We say that a Turing Machine *solves* a computational problem $f : \Sigma^* \to 2^{(\Gamma/\sqcup)^*}$ if the Turing machine which starts with input x halts with some output in $f(x)$ as the set of characters before the first $\sqcup$.

The **extended Church-Turing Thesis** says that every reasonable model of computation can be simulated on a Turing machine with only a polynomial slowdown.

Exercise. *Give the state diagram for a Turing machine which converts unary to binary (that is, given the input 1^x, it should return the binary representation of x).*

Solution.
We give a two-tape Turing machine, which does the following:

- Adds a special start symbol \$ to the beginning of the tape and shifts the input 1 character right ($q_0 \to q_S \to \ldots \to q_S \to q_1$).
- Going left until it hits the \$, counts the number of 1's by incrementing on the second tape, keeping the answer written in reverse on the second tape ($q_1 \to q_2 \to q_3 \to q_1$). Each time it returns to q_1, the head of the second tape is at the first character after the \$.
- It removes the \$ at the start of the first tape and scrolls the head of the second tape to the end of its string ($q_1 \to q_m \to q_r$).
- It copies the second tape backwards onto the first tape, halting when it reaches the \$ at the start of the second tape ($q_r \to q_{halt}$).



The runtime of this machine is $O(n \log n)$, where n is the length of the input. Note that in the diagram above, we have omitted "impossible" arrows, e.g. ones that the machine will never follow given a valid input.

Often, when we talk about computational problems, we talk about decision problems, where the output is in $\{0, 1\}$; this is the same idea as deciding whether the given string is in some language $L \in \Sigma^*$.

However, in many cases the problem of finding a solution is not necessarily more "difficult" than the problem of determining whether there exists one. You're asked to prove a few of these on your homework; we'll look at one more of them here.

Exercise. *Show that there is a polynomial-time algorithm for* 3-SATISFIABILITY *(given a boolean formula in 3-CNF form, determine whether there exists a satisfying assignment) if and only if the computational problem of finding a satisfying assignment (or returning nothing if none exists) is in P.*

For this problem, you can use ***high-level*** *algorithm descriptions.*

Solution.
($\Longleftarrow$) Clearly, if we can find a satisfying assignment, then we can determine whether there exists one.

($\Longrightarrow$) Now suppose we have an algorithm A that decides in polynomial time whether a given 3-CNF formula ϕ has a satisfying assignment.

How can we use A to find such an assignment? The idea is to set the variables one by one, and make sure that the remaining formula still has a satisfying assignment along the way.

Algorithm: So, let $\phi_{i_1,\ldots,i_t}$ stand for the formula ϕ with partial assignment $x_1 = i_1, \ldots, x_t = i_t$; then $\phi_{i_1,\ldots,i_t}$ can easily be written in 3-CNF form (why? there is a detailed explanation in the more formal solution given below). If $A(\phi)$ returns FALSE, we return nothing. Otherwise, we run $A(\phi_1)$. If that gives

TRUE, we assign $x_1 = 1$ and repeat with the formula ϕ_1 instead of ϕ; if it gives FALSE, it must be the case that $A(\phi_0)$ is TRUE, so we repeat with ϕ_0 instead of ϕ. In the end we have a formula $\phi_{i_1,\ldots,i_n}$ which is satisfiable, but is also a constant, therefore $x_j = i_j$ gives a satisfying assignment.

Correctness and running time: From the above discussion, correctness is clear. The running time will be $O(nT(n))$ where $T(n)$ is the running time of the decider for satisfiability, hence it is polynomial.

More detail: TM implementation. Here's a more low-level solution to give you more practice with TMs:

Suppose that we have some Turing Machine M, which, given an input, determines whether there exists a satisfying assignment. We go about actually finding a satisfying assignment as follows:

- Simulate M on the input (after copying the input to a second tape); if M returns false, then halt and return nothing.
- Now, try setting $x_1 = 1$ in the formula, e.g. copy the formula onto a second tape, removing any clauses which are satisfied (i.e. contain x_1) and removing any instances of $\neg x_1$. If the latter results in any empty clauses, e.g. saying that this is not a satisfying assignment, set $x_1 = 0$.
- Run M on the output of the previous step; if M returns true, then set $x_1 = 1$ in the original copy of the formula. Otherwise, set $x_1 = 0$.
- Repeat the previous two steps with $x_2, \ldots, x_n$ in sequence. The final result will be a satisfying solution.

Here, we see that we simulate M at most n times, and that the length of the input to M is no more than the length of the input to our original problem. Meanwhile, all the other steps (copying the formula, setting the value of variable) take no more than linear time and run only polynomially many times, so they also run in polynomial time. Hence, this entire algorithm runs in polynomial time.

There are many details that we don't address in the solution above because we are working at a higher level (e.g. clearing a tape so that we can reuse it in multiple simulations of M); however, you should convince yourself that all of these are doable, and none of these have a significant affect on the runtime.