CSE491/596 Lecture Wed. 10/4/23: Simulation Theorems

Before we take a second look at the "Universal RAM Simulator", let's see another example of a Read-Evaluate-Write loop. (Called a REPL in programming languages.) First, a definition:

Definition: A Turing machine M **runs in time $t(n)$** if for all n and inputs x of length n, $M(x)$ halts within $t(n)$ steps. If M is nondeterministic, all possible computations must halt within $t(n)$ steps.

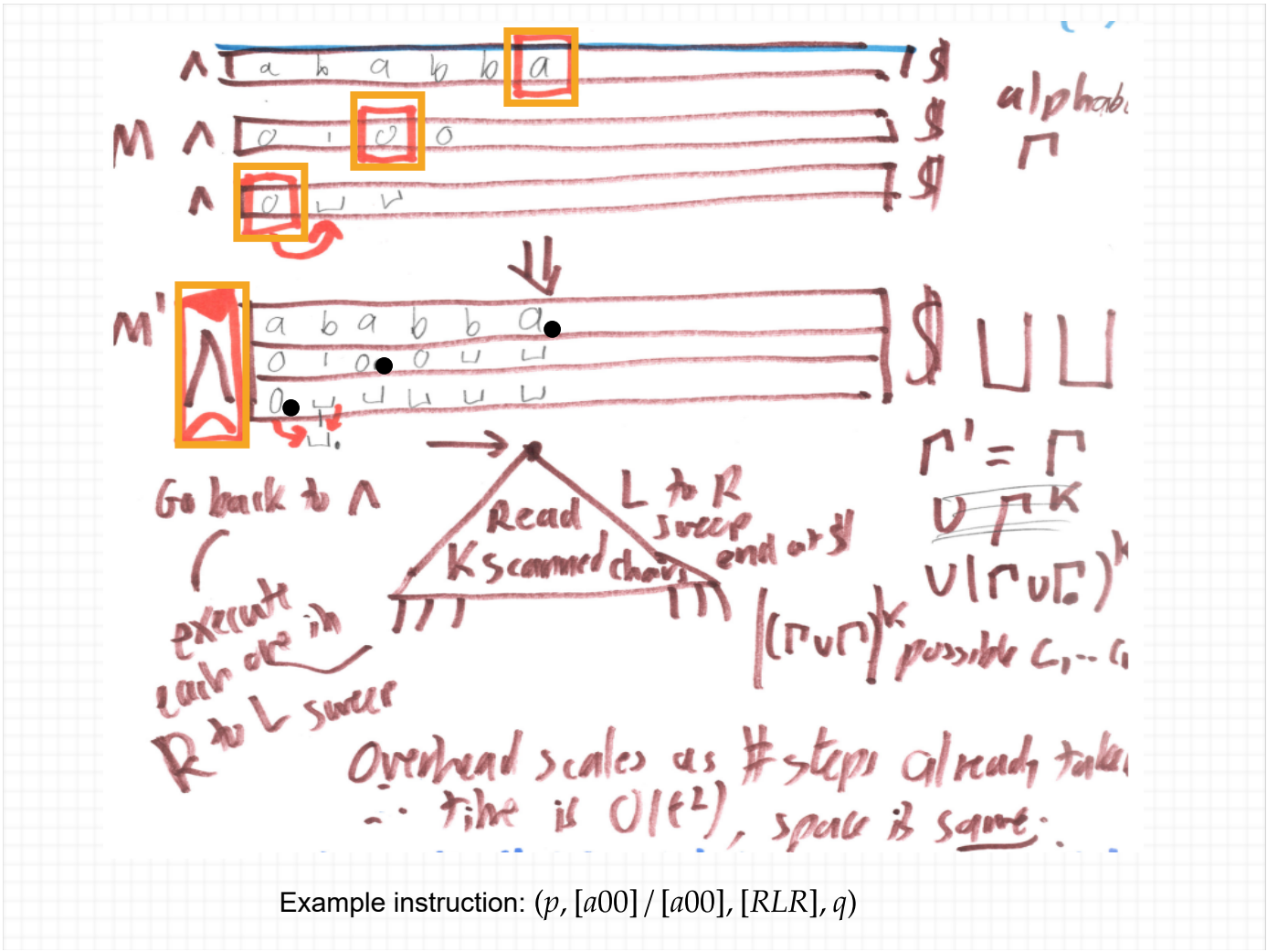
For example, every DFA---and every NFA without ϵ-transitions---runs in time $t(n) = n + 1$, which is the fastest possible time that reads every input char and the blank that says the input is terminated. (This is sometimes called running in **real time**.) It is convenient to apply O-notation to time without caring about the exact number of steps.

Multi-Tapes to Single Tapes

All the 2-tape machines we have seen have run in $O(n)$ time, which is called **linear time**, but some of the 1-tape machines have run in $\Theta(n^2)$ time, which is **quadratic time**. For some languages that are in linear time on 2-tape TMs, such as $PAL = \{x : x = x^R\}$, where x^R means x reversed and "$x = x^R$" defines a **palindrome**, one can prove that single-tape TMs cannot do better than quadratic time. But at least they can't do worse:

Theorem 1: For any k-tape TM $M = (Q, \Sigma, \Gamma, \delta, \sqcup, s, F)$ that runs in time $t(n)$, we can build a 1-tape TM M' that simulates M and runs in $O(t(n)^2)$ time.

Proof Sketch: M' uses work alphabet $\Gamma' = \Gamma^k$, which can pack the k chars in any "column j" of the k tapes of M into one "superchar" in cell j on the one tape of M'. We also need chars that say whether they are currently being scanned by a tape head of M, so we actually have $\Gamma' = (\Gamma \cup \Gamma_{\odot})^k$ where $\Gamma_{\odot}$ is a "dotted copy" of Γ. Here is a diagram of how the memory map of M' relates to M for $k = 3$, which is the number of tapes in our "Universal Ram Simulator":



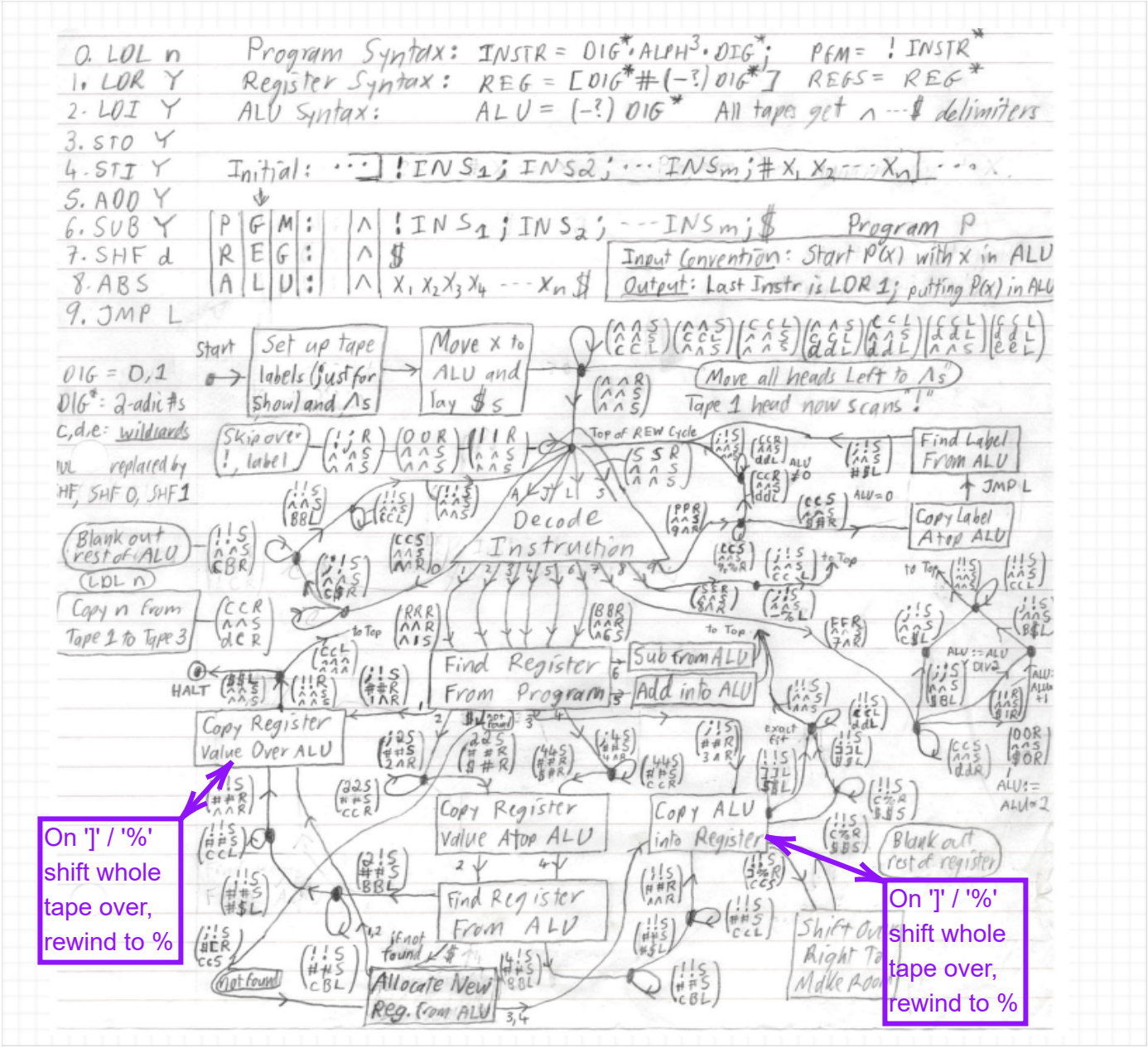
Example instruction: $(p, [a00] / [a00], [RLR], q)$

Initially, $M'(x)$ converts each char x_i into the "superchar" $[x_i \_ \_ \_ \dots \_]$ which packs x_i and $k-1$ blanks into one char of Γ and rewinds its single tape head onto the superchar $[\ \wedge\ \odot\_\odot\_\odot\ \cdots\ \_\odot]$ which lines up the k "virtual" tape heads of M on $\wedge$ and blanks below it to the left of x. Thereafter, M' simulates each step of M in one *left-to-right* **pass** that reads the k-tuple of scanned characters according to which parts of superchars have $\odot$ and then a *right-to-left* pass that performs the corresponding instruction of δ. This leaves M' ready to simulate the next step by M.

The total time for each pass is initially $2n+4$ but can grow if and when M uses more tape cells beyond the end(s) of x. The width of a pass cannot be more than (twice the) time taken by M thus far, so it is always less than $t(n)$ (or less than $2t(n)$, if M uses cells to the left of x as well). Thus the total time is $O\big(t(n)^2\big)$. ⊠

Universal Simulation in Polynomial Time

Now let's see how much time overhead the "Universal RAM Simulator" has:



The following attempt at detailed time analysis can be taken as FYI:

- Let us suppose the given program P is fixed, and let n be the length of whatever input x is given to the program.
- The RAM can make multiple copies of x and allocate new registers to store them. So in t steps by the RAM, the register tape can swell to size $\sim nt$. It cannot grow more because we did not give the RAM a MUL instruction---or any instruction that can double the side of the operand. (You have to multiply by repeated addition. Cf. **RISC**.)
- Once you have order-of t size-n registers on the tape, it can take order-of nt steps to look up any one. Over t sequential steps, this can take $\Theta(nt^2)$ total time...
- ...except that storing a slightly bigger value to a register can exhaust its room. This is signaled by the Turing machine getting the register's closing `']'` character when it was still wanting to write

another character from the ALU tape. The TM needs to "make room" by ***shifting the entire register tape from that point rightward one more cell to the right.***

- This can employ the shift-over routine given in machine detail in Monday's lecture, say using `'%'` to represent the fresh blank register cell rather than $\wedge$. A separate copy of that code needs to be attached everywhere M is storing to a register, but that's OK: the original M diagram is finite so only finitely many copies of the code for this "shift-over" **daemon** need to be added.
- The time to run the daemon is order-nt. Presuming that the RAM word size stays no larger than $O(n)$, you need it at most $O(n)$ times per each step of the RAM, for $O(n^2\ t)$ time shifting overall. (The RAM word size can grow, but via an **amortized analysis** that takes into account the time it takes the RAM overall to grow it, this assumption stays roughly valid.) This dominates the $O(nt)$ steps to look up the register.
- So over t steps by the RAM, the worst-case time for M is $O(n^2\ t^2)$ time.

If we suppose $t \geq n$ then this is $O(t^4)$ time. Doing a mega-handwave now, if you employ a caching scheme on the register tape analogous to the C++ vector object works, you can bring this down to $O(t^3 \log t)$ time. Steve Cook, with some work joint with his student Robert Reckhow at the University of Toronto, proved this and also that if the RAM uses the "**fair cost**" time measure (by which the cost of a basic operation is the number of bits in its operands), then the time overhead for $O(t)$ fair-cost RAM time on inputs of length n is $O(t^2 \log n)$ time by the TM.

Couple with the quadratic time overhead of 3-tapes-to-1, this translates to saying that t steps on the RAM can be simulated by from $O(t^8)$ to $\widetilde{O}(t^4)$ time by the single-tape TM, depending on how one regards the RAM time and whether you make more-complicated code via caching. **In all events, it is a polynomial time overhead.** That enables us to state the following theorem:

Theorem 2: For every program P written in any known executable programming language $\mathscr{L}$ (high-level or otherwise) that uses standard input and standard output, we can build a 3-tape Turing machine M_P such that whenever P given x on standard input writes y to standard output, M_P given x on its input tape writes y to a special output tape. If $P(x)$ halts, then $M_P(x)$ halts---and if $P(x)$ halts within t steps, then $M_P(x)$ halts within $t^{O(1)}$ steps.

Proof: First, any compiler for $\mathscr{L}$ to a known code target can be converted into a compiler from $\mathscr{L}$ to the "mini-assembler"---which is essentially similar to what the text calls a RAM. So we can compile P to make an equivalent RAM program R_P. Then take M_P to be the Turing machine T in the handout, but with the binary text of R_P already written on its input tape. More precisely, M_P begins with a series of dedicated instructions that write out R_P char-by-char in front of any input x on its first tape, so it has $R_P\#x$ there. Then it just segues to the start state of T. $\boxtimes$

Theorem 2: We can build a **universal Turing machine**, meaning a single TM U that takes inputs of the form $\langle M, x \rangle$ and simulates $M(x)$, again with polynomial-time overhead.

Here $\langle M, x \rangle$ denotes an unspecified but transparent way of combining the code of M and the bits of x into a single string over whatever alphabet we need. In the Turing Kit, user-designed Turing machines M are stored as ASCII files, so that can be the code $\langle M \rangle$ of M. ASCII can be converted to strings over $\{0, 1\}$ if we so desire. The files are self-delimiting, so we can then define $\langle M, x \rangle$ by just appending x to $\langle M \rangle$. Or, assuming that neither M nor x has any commas or angle brackets, we can regard $\langle M, x \rangle$ as literally '⟨' then whatever string code of M, then comma, then x, and finally '⟩'. The choice of **tupling scheme** does not matter in detail.

Proof: The *Turing Kit* is a high-level Java program P that reads a TM M and an input x and executes $M(x)$. That is (essentially), $P(\langle M, x \rangle) = M(x)$. Then compile P to M_P as above and call it U. Then $U(\langle M, x \rangle) = P(\langle M, x \rangle) = M(x)$. This notation includes that $U(\langle M, x \rangle) \downarrow$ if and only if $M(x) \downarrow$. (The down arrow means "halts" while $\uparrow$ is read as "diverges" or "does not halt.") ☒

The import is, simply: **Turing machines have the same computing power as high-level programming languages, likewise the same power as the machines on which they run**. This is the main concrete evidence in support of the following. Alonzo Church had earlier defined notions of "recursive" and "r.e." via logical schemes of recursion, before Alan Turing's famous 1936 paper proved his machines equivalent to them. Church became Turing's PhD advisor at Princeton in 1937--38; I met him when he received an honorary doctorate from UB in 1990.

The Church-Turing Thesis (three-part version):

1. Any HLL that will ever be devised will have the same computing power as the Turing machine.
2. Any physical device that will ever be built---even quantum computers---will have no more computing power than a Turing machine.
3. For any human being H who follows a consistent functional procedure to convert (sensory) inputs x into outputs y, there exists a Turing machine M_H that on the same inputs x (under a natural string encoding, e.g., pixels for optical input) outputs the same values y. Moreover, M_H has comparable program size and efficiency to the "grey matter" of H, or better.

Plank 1 is often considered a "truism" but maybe it depends on plank 2, which survived a "quantum scare" from David Deutsch at Oxford in 1985 and is even more in play when we bring time-efficiency into the picture. Plank 3 is the philosophically controversial one; the program and memory size S needed is the threshold that "The Singularity" talks about. The "Part Deux" of the C-T thesis is often ascribed to Alan Cobham and Jack Edmonds from papers they wrote in 1965, in which they justified **polynomial time** as a benchmark for feasible problem-solving.

Polynomial-Time C-T Thesis: As above, plus the assertion that whatever the HLL and/or device physically implementing its programs, there will always be a constant k such that whatever the program/device does in time t can be emulated by $O(t^k)$ steps of the Turing machine.

This was also almost-universally believed until 1994, when Peter Shor proved that quantum computers can factor n-digit numbers in $\widetilde{O}(n^2)$ time (idealized---no one has yet built quantum technology that can *scale up*), whereas the security of most Internet commerce and many other cryptosystems relies on concrete scaling of the belief that factoring requires roughly $2^{\Omega(n^{1/3})}$ time, well maybe $2^{\Omega(n^{1/4})}$ or $2^{\Omega(n^{1/5})}$ time in most cases... [Cf. the 1992 movie *Sneakers* and the novel *Factor Man*.]

But as long as we stick with "classical" machines---meaning non-quantum hardware---we can take both theses as given. (Note: Actually, transistors and other chip elements *are* quantum devices, but the point is that they treat information in the classical manner of *bits*, as opposed to *qubits*.) The import is:

The classes REC, RE, *and co*-RE, *and later* P, NP, *and co*-NP, *remain the same whenever we transfer their defining notions to any HLL or classical machine model. Moreover, it is perfectly legitimate to describe Turing machines via pseudocode, provided the pseudocode gives enough detail to pin down the running time t within a linear $O(t)$, a quasi-linear $\widetilde{O}(t)$, or at worst a polynomial $t^{O(1)}$, factor.*

For example, the 2-tape TM we built to recognize $\{a^m b^n : m = n\}$ can be described by saying, "Copy leading a's to tape 2, then count against b's on the rest of tape 1, and accept iff the counts are equal and the end is reached on tape 1 without any further a appearing. Runtime: $O(m+n)$ steps, which is linear in the length $m+n$ of the input."

A Simulation Not in Polynomial Time

Theorem 3: For every nondeterministic TM N we can build a deterministic TM M such that $L(M) = L(N)$.

Proof: The Turing Kit could be upgraded to a version T' that simulates a given NTM N on an input x by branching to try all possibilities, accepting if and when some branch accepts x. The program T' itself is deterministic. Hence so is the equivalent Turing machine $M_{T'}$ obtained from T' via Theorem 1. ☒

The one thing we don't know how to do is make T' avoid exponential branching, which slows down the time exponentially. This is different from the situation with an NFA N on a given input x, where we can simulate $N(x)$ by the trick of maintaining the current set R_i of possible states after each bit i of x, and thus avoid the exponential blowup of converting N into a DFA. Whether we can do a similar trick for a general NTM N is the infamous NP = ? P problem, which we will come to soon.