Notes on Complexity Theory Last updated: August, 2011

Lecture 1

Jonathan Katz

1 Turing Machines

I assume that most students have encountered Turing machines before. (Students who have not may want to look at Sipser's book [3].) A Turing machine is defined by an integer $k \geq 1$, a finite set of states Q, an alphabet Γ, and a transition function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, S, R\}^k$ where:

- k is the number of (infinite, one-dimensional) tapes used by the machine. In the general case we have $k \geq 3$ and the first tape is a read-only *input tape*, the last is a write-once *output tape*, and the remaining $k-2$ tapes are *work tapes*. For Turing machines with boolean output (which is what we will mostly be concerned with in this course), an output tape is unnecessary since the output can be encoded into the final state of the Turing machine when it halts.
- Q is assumed to contain a designated start state q_s and a designated halt state q_h. (In the case where there is no output tape, there are two halting states $q_{h,0}$ and $q_{h,1}$.)
- We assume that Γ contains $\{0, 1\}$, a "blank symbol", and a "start symbol".
- There are several possible conventions for what happens when a head on some tape tries to move left when it is already in the left-most position, and we are agnostic on this point. (Anyway, by our convention, below, that the left-most cell of each tape is "marked" there is really no reason for this to ever occur...).

The computation of a Turing machine M on input $x \in \{0, 1\}^*$ proceeds as follows: All tapes of the Turing machine contain the start symbol followed by blank symbols, with the exception of the input tape which contains the start symbol followed by x (and then the remainder of the input tape is filled with blank symbols). The machine starts in state $q = q_s$ with its k heads at the left-most position of each tape. Then, until q is a halt state, repeat the following:

1. Let the current contents of the cells being scanned by the k heads be $\gamma_1, \ldots, \gamma_k \in \Gamma$.
2. Compute $\delta(q, \gamma_1, \ldots, \gamma_k) = (q', \gamma'_2, \ldots, \gamma'_k, D_1, \ldots, D_k)$ where $q' \in Q$ and $\gamma'_2, \ldots, \gamma'_k \in \Gamma$ and $D_i \in \{L, S, R\}$.
3. Overwrite the contents of the currently scanned cell on tape i to γ'_i for $2 \leq i \leq k$; move head i to the left, to the same position, or to the right depending on whether $D_i = L, S$, or R, respectively; and then set the current state to $q = q'$.

The output of M on input x, denoted $M(x)$, is the binary string contained on the output tape (between the initial start symbol and the trailing blank symbols) when the machine halts. (When there is no output tape, then the output is '1' if M halts in state $q_{h,1}$ and the output is '0' is M halts in state $q_{h,0}$.) It is also possible that M never halts when run on some input x. We return to this point later.

The *running time* of a Turing machine M on input x is simply the number of "steps" M takes before it halts; that is, the number of iterations (equivalently, the number of times δ is computed) in the above loop. Machine M is said to run in time $T(\cdot)$ if for every input x the running time of $M(x)$ is at most $T(|x|)$. The *space* used by M on input x is the number of cells written to by M on all its *work* tapes[1] (a cell that is written to multiple times is only counted once); M is said to use space $T(\cdot)$ if for every input x the space used during the computation of $M(x)$ is at most $T(|x|)$. We remark that these time and space measures are *worst-case* notions; i.e., even if M runs in time $T(n)$ for only a fraction of the inputs of length n (and uses less time for all other inputs of length n), the running time of M is still said to be T. (Average-case notions of complexity have also been considered, but are somewhat more difficult to reason about.)

A Turing machine M *computes a function* $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all x. Assuming f is a total function, and so is defined on all inputs, this in particular means that M halts on all inputs. We will focus most of our attention on boolean functions, a context in which it is more convenient to phrase computation in terms of *languages*. A language is simply a subset of $\{0,1\}^*$. There is a natural correspondence between languages and boolean functions: for any boolean function f we may define the corresponding language L as the set $L = \{x \mid f(x) = 1\}$. Conversely, for any language L we can define the boolean function f so that $f(x) = 1$ iff $x \in L$. A Turing machine M *decides a language* L if

$$x \in L \Leftrightarrow M(x) = 1$$

(we sometimes also say that M *accepts* L, though we will try to be careful); this is the same as computing the boolean function f that corresponds to L.

1.1 Comments on the Model

Turing machines are *not* meant as a model of modern computer systems. Rather, they were introduced (before computers were even built!) as a mathematical model of *what computation is*. Explicitly, the axiom is that "any function that can be computed in the physical world, can be computed by a Turing machine"; this is the so-called *Church-Turing thesis*. (The thesis cannot be proved unless one can formally define what it means to "compute a function in the physical world" without reference to Turing machines. In fact, several alternate notions of computation have been defined and shown to be equivalent to computation by a Turing machine; there are no serious candidates for alternate notions of computation that are *not* equivalent to computation by a Turing machine. See [1] for further discussion.) In fact, an even stronger axiom known as the *strong Church-Turing thesis* is sometimes assumed to hold: this says that "any function that can be computed in the physical world, can be computed with at most a polynomial reduction in efficiency by a Turing machine". This thesis is challenged by notions of *randomized* computation that we will discuss later. In the past 15 years or so, however, this axiom has been called into question by results on *quantum computing* that show polynomial-time algorithms in a quantum model of computation for problems not known to have polynomial-time algorithms in the classical setting.

There are several variant definitions of Turing machines that are often considered; none of these contradict the strong Church-Turing thesis. (That is, any function that can be computed on any of these variant Turing machines, including the variant defined earlier, can be computed on any other

[1] Note that we do not count the space used on the input or output tapes; this allows us to meaningfully speak of sub-linear space machines (with linear- or superlinear-length output).

variant with at most a polynomial increase in time/space.) Without being exhaustive, we list some examples (see [1, 2] for more):

- One may fix Γ to *only* include $\{0, 1\}$ and a blank symbol.
- One may restrict the tape heads to only moving left or right, not staying in place.
- One may fix $k = 3$, so that there is only one work tape. In fact, one may even consider $k = 1$ so that there is only a single tape that serves as input tape, work tape, and output tape.
- One can allow the tapes to be infinite in both directions, or two-dimensional.
- One can allow random access to the work tapes (so that the contents of the ith cell of some tape can be read in one step). This gives a model of computation that fairly closely matches real-world computer systems, at least at an algorithmic level.

The upshot of all of this is that it does not matter much which model one uses, as long as one is ok with losing polynomial factors. On the other hand, if one is concerned about "low level" time/space complexities then it is important to fix the exact model of computation under discussion. For example, the problem of deciding whether an input string is a palindrome can be solved in time $O(n)$ on a two-tape Turing machine, but requires time $\Omega(n^2)$ on a one-tape Turing machine.

1.2 Universal Turing Machines and Uncomputable Functions

An important observation (one that is, perhaps, obvious nowadays but was revolutionary in its time) is that *Turing machines can be represented by binary strings.* In other words, we can view a "program" (i.e., a Turing machine) equally well as "data", and run one Turing machine on (a description of) another. As a powerful example, a *universal* Turing machine is one that can be used to simulate any other Turing machine. We define this next.

Fix some representation of Turing machines by binary strings, and assume for simplicity that every binary string represents some Turing machine (this is easy to achieve by mapping badly formed strings to some fixed Turing machine). Consider the function $f(M, x) = M(x)$. Is f computable?

> **Note:** Here f is a partial function, since in this context the given Turing machine M may not halt on the given input x and we leave f undefined in that case. A partial function f is computable if there is a Turing machine U such that for all x where f is defined we have $U(x) = f(x)$. When $f(x)$ is undefined the behavior of U may be arbitrary. An alternative is to consider the (total) function
>
> $$f'(M, x, 1^t) = \begin{cases} 1 & \text{if } M(x) \text{ halts within } t \text{ steps with output } 1 \\ 0 & \text{otherwise} \end{cases},$$
>
> whose computability is closely linked to that of f. Another natural possibility is to consider the (total) function
>
> $$f_{halt}(M, x) = \begin{cases} 1 & \text{if } M(x) \text{ halts with output } 1 \\ 0 & \text{otherwise} \end{cases};$$
>
> as we will see, however, f_{halt} is *not* computable.

Perhaps surprisingly, f is computable. We stress that here we require there to be a *fixed* Turing machine U, with a fixed number of tapes and a fixed alphabet (not to mention a fixed set of states) that can simulate the behavior of an *arbitrary* Turing machine M that may use any number of tapes and any size alphabet. A Turing machine computing f is called a *universal* Turing machine.

Theorem 1 *There exists a Turing machine U such that (1) $U(M, x) = M(x)$ for all x for which $M(x)$ is defined; furthermore, (2) for every M there exists a constant c such that the following holds: for all x, if $M(x)$ halts within T steps, then $U(M, x)$ halts within $c \cdot T \log T$ steps.*

Proof We only sketch the proof here. We consider the case where M computes a boolean function, and so has no output tape; U will not have an output tape either. U will use 3 work tapes, and the alphabet Γ that only includes $\{0, 1\}$, a blank symbol, and a start symbol. At a high level, U proceeds as follows:

1. First, U applies a transformation to M that results in a description of an equivalent machine M' that uses only a single work tape (in addition to its input tape). This is known to be possible, and moreover is possible in such a way that the following holds: if $M(x)$ halts within T steps, then $M'(x)$ halts within $O(T \log T)$ steps (see [2, Chap. 12] or [1, Sect. 17]). The description of M' is stored on the second work tape of U (the remaining work tapes of U are used to perform the transformation).

2. Next, U applies a transformation to M' that results in a description of an equivalent machine M'' that uses the binary alphabet (plus blank and start symbol). This is known to be possible with only a constant-factor loss of efficiency (see [1, 2]). Thus if $M(x)$ halts within T steps, then $M''(x)$ halts within $O(T \log T)$ steps. The description of M'' is stored on the first work tape of U (the 3rd work tape of U can be used to perform the transformation).

3. Finally, U simulates the execution of M'' on input x. It can do this by recording the current state of M'' on its second work tape (recall that the description of M'' itself is stored on the first work tape of U) and using its third work tape to store the contents of the work tape of M''. To simulate each step of M'', we have U simply scan the entire description of M'' until it finds a transition rule matching the current state of M'', the current value being scanned in the input x, and the current value being scanned in the work tape of M''. This rule is then used by U to update the recorded state of M'', to move its heads on its input tape and third work tape (which is storing the work tape of M''), and to rewrite the value being scanned on the work tape of M''. If M'' halts, then U simply needs to check whether the final state of M'' was an accepting state or a rejecting state, and move into the appropriate halting state of its own.

It is not hard to see that $U(M, x) = M(x)$ for any x for which $M(x)$ halts. As for the claim about the running time, we note the following: the first and second steps of U take time that depends on M but is *independent* of x. In the third step of U, each step of M'' is simulated using some number of steps that depends on M'' (and hence M) but is again independent of x. We have noted already that if $M(x)$ halts in T steps then $M''(x)$ halts in $c'' \cdot T \log T$ steps for some constant c'' that depends on M but not on x. Thus $U(M, x)$ halts in $c \cdot T \log T$ steps for some constant c that depends on M but not on x. ■

We have shown that (the partial function) f is computable. What about (the function) f_{halt}? By again viewing Turing machines as data, we can show that this function is not computable.

Theorem 2 *The function f_{halt} is not computable.*

Proof Say there is some Turing machine M_{halt} computing f_{halt}. Then we can define the following machine M^*:

> On input (a description of) a Turing machine M, output $M_{halt}(M, M)$. If the result is 1, output 0; otherwise output 1.

What happens when we run M^* on *itself*? Consider the possibilities for the result $M^*(M^*)$:

- Say $M^*(M^*) = 1$. This implies that $M_{halt}(M^*, M^*) = 0$. But that means that $M^*(M^*)$ does not halt with output 1, a contradiction.
- Say $M^*(M^*) = 0$. This implies that $M_{halt}(M^*, M^*) = 1$. But that means that $M^*(M^*)$ halts with output 1, a contradiction.
- It is not possible for $M^*(M^*)$ to never halt, since $M_{halt}(M^*, M^*)$ is a total function (and so is supposed to halt on all inputs).

We have reached a contradiction in all cases, implying that M_{halt} as described cannot exist. ■

Remark: The fact that f_{halt} is not computable does *not* mean that the halting problem cannot be solved "in practice". In fact, checking termination of programs is done all the time in industry. Of course, they are not using algorithms that are solving the halting problem – this would be impossible! Rather, they use programs that may give *false negative*, i.e., that may claim that some other program does not halt when it actually does. The reason this tends to work in practice is that the programs that people want to reason about in practice tend to have a form that makes them amenable to analysis.

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[3] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.