Lecture 14: Time Complexity

October 25, 2016
CS 1010 Theory of Computation

When we talk about time complexity, we limit Turing machines to deciders. Otherwise, a machine might not halt, so we don’t have a well-defined notion of running time.

Topics Covered

1. Time Complexity
2. Runtimes of TM Variants
3. The Class P

1 Time Complexity

• Let $M$ be a decider. The running time (or “runtime” or “time complexity”) of $M$ is the function $f(n)$ defined to be the maximum number of steps $M$ takes to process an input of length $n$. We say that $M$ runs in time $f(n)$, or that $M$ is an $f(n)$-time TM.

• For an NTM $N$, the runtime of $N$ is the maximum number of steps any branch of computation of $N$ takes on any input of length $n$. As long as the NTM is a decider, every branch takes finite time.

We can consider running time in terms of function families; in particular, using Big O notation. Recall three types of runtime bounds:

1. $f(n) = O(g(n))$ if there exist constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
2. $f(n) = \Omega(g(n))$ if there exist constants $c$ and $n_0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.
3. $f(n) = \Theta(g(n))$ if $f = O(g(n))$ and $f = \Omega(g(n))$.

As an example, consider the language $L_{\text{repeat}} = \{w\#w\}$. Here is a TM $M$ that decides $L_{\text{repeat}}$:
$M$, on input $x$:

1. Verify that $x = y \# z$.
2. Keep marking symbols in $y$ from left to right, and match them with symbols in $z$. Also mark the matched symbols in $z$.
3. If $y$ and $z$ fail to match, reject. Otherwise, accept.

How long does it take $M$ to process a string of length $n$? $M$ scans its input for approximately $n$ steps for each of approximately $\frac{n}{2}$ symbols in $y$. The runtime is thus quadratic, $\Theta(n^2)$. It turns out that this runtime is optimal, meaning that there cannot be a standard TM that decides $L\text{\_\text{repeat}}$ in less than quadratic time.

**Proof Idea that Quadratic is Optimal**  Consider a string of the form $u \# v \# 0^k$, where $|u| = |v| = k$. This string is in $L\text{\_\text{repeat}}$ if $u = v$. Suppose that we have “snapshots” of the state the TM is in when it reaches a specific cell in the first $0^k$ substring. Let the states be $q_{i_1}, q_{i_2}, \ldots, q_{i_n}$. We also store the contents of the cell when a state is reached. Suppose that given the string $u$, we could use the snapshots to decide if $v = u$. In this case, the snapshots must contain $k$ bits of information. A single cell contains 1 bit of information, so we must visit this cell a linear number of times with respect to $k$. This holds for all tape locations in the first $0^k$ substring, so at least $k \cdot k = k^2$ steps must be made. This is not a full proof, but it is the general idea used to prove that a standard TM can decide $L\text{\_\text{repeat}}$ in at best quadratic time.

Alternatively, suppose we decide $L\text{\_\text{repeat}}$ with a two-tape TM $M$. Consider the following algorithm:

$M$, on input $x$:

1. Verify that $x = y \# z$.
2. As you read $y$, copy $y$ onto tape 2.
3. Return the second head to the start of tape 2.
4. As you read $z$, match symbols of $y$ and $z$.
5. Reject if any symbols don’t match. Otherwise, accept.
This algorithm takes linear time, as \( M \) scans back and forth a constant number of times. That is, while a single-tape TM takes quadratic time, a multi-tape TM takes linear time. We can generalize this relationship between the runtimes of single-tape and multi-tape TMs.

2 Runtimes of TM Variants

**Theorem** Every \( t(n) \)-time multi-tape TM has an equivalent \( O(t^2(n)) \)-time standard TM.

**Proof Idea** Recall the algorithm for constructing an equivalent single-tape TM from a multi-tape TM (see Lectures 9 and 10). A general outline of the algorithm is:

1. Represent the tapes of the multi-tape TM on a single tape, separated by special symbols.
2. For every tape, mark the location of the tape’s head.
3. At every iteration, advance the head on one of the tapes as follows:
   (a) Scan the whole tape.
   (b) Determine the next transition, and make it.
   (c) Possibly free up extra room.

Consider the amount of time one step of multi-tape TM takes on a standard (or “canonical”) TM. Step (a) requires looking at every cell on the active portions of each of the multi-tape TM’s tapes. Each of these active portions has length at most \( t(n) \), as a TM cannot process more than \( t(n) \) cells in \( t(n) \) time. To scan all of these tapes, a single-tape TM takes at most a constant multiple of \( t(n) \) steps, so the runtime of step (a) is \( O(t(n)) \). In total, the multi-tape TM takes \( t(n) \) steps. The single-tape TM simulates all of these, each one taking time \( O(t(n)) \), so the total runtime is \( O(t^2(n)) \).

- The complexity class \( \text{TIME}(t(n)) = \{ L | L = L(M) \text{ and } M’s \text{ runtime is } O(t(n)) \} \).

For example, \( L_{\text{repeat}} \in \text{TIME}(n^2) \). If \( t(n) = O(g(n)) \), then \( \text{TIME}(t(n)) \subseteq \text{TIME}(g(n)) \).

**Theorem** Let \( t(n) \geq n \). Then every \( t(n) \)-time NTM has an equivalent \( 2^{O(t(n))} \)-time standard TM.
Proof We know that as a model of computation, NTMs are equivalent to TMs. Recall the process to convert an NTM $N$ to a standard TM $M$:

$M$, on input $x$:

1. For $k$ from 0 to $\infty$:
   (a) Let $u_1 \ldots u_k$ be the next string in lexicographic order over the alphabet $\{1, \ldots, d\}$ where $d$ is the maximum number of transitions from a single $(q, a)$ input.
   (b) Simulate $N$ on input $x$, using $u_1 \ldots u_k$ as the nondeterministic choices.
   (c) Accept if $N$ accepts.
   (d) If $N$ has rejected each time for this $k$, reject.
   (e) Else, go back to (a) and increment $k$. Try the next set of nondeterministic choices.

This algorithm takes at most $d^{O(n)}$ steps, as we might explore every possible branch of computation. Note that we can multiply by a constant to change the base of the power from $d$ to 2. As a result, the runtime can be expressed as $2^{O(t(n))}$. ■

3 The Class P

- The class $P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$.

A language $L$ is in P if $L \in \text{TIME}(n^k)$ for some $k$; that is, $L$ can be decided by an $O(n^k)$-time TM. We can also say that there is a TM that decides $L$ in polynomial time ("poly-time").

One of the reasons this is a useful classification of languages is that it is model-independent. A poly-time multi-tape TM has an equivalent poly-time single-tape TM. Thus, the class P is the same regardless of the number of tapes of the TM. P is also the same when considering the Random Access Model (RAM). A RAM machine can be modeled by a TM with two tapes: a work and memory tape, and an address tape.

Many people consider P to be a good measure of computations that can realistically be done with computer programs. While something that takes $O(n^{100})$ is technically polynomial, smaller polynomials capture problems that require a realistic amount of time to compute.

The following languages are examples of languages in P:
1. PATH = \{\langle G, s, t \rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t \}\)

2. RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are binary-encoded integers such that } \gcd(x, y) = 1 \}\)

3. Any language that is context-free

**Poly-time Algorithm for RELPRIME**  On input \langle x, y \rangle, we can compute the greatest common divisor of \(x\) and \(y\) using the Euclidean algorithm, and then accept if the result is 1. In the Euclidean algorithm, at each step, you either halve the magnitude of \(x\), or you will halve its magnitude in the next step. The number of iterations is thus a constant multiple of the number of bits in \(x\). Overall, then, the runtime is polynomial with respect to the size of \langle x, y \rangle.

**Poly-time Algorithm for a CFL**  We can determine membership in a CFL using a dynamic program. Let \(L\) be a CFL, and let \(G\) be its grammar in Chomsky normal form. Consider the following algorithm:

```
On input \(w = w_1 \ldots w_n\):

1. Make an \(n \times n\) table. Cell \((i, j)\) will contain all variables in \(G\) that derive \(w_i \ldots w_j\), where \(j \geq i\). If \(j < i\), the cell can contain any other nonsense that we won’t use.

2. First, fill in the diagonal by finding variables that generate terminals.

3. For \(k\) from 1 to \(n - 1\), for \(i\) from 1 to \(n\), for \(j\) from 1 to \(k - 1\):
   
   (a) Let \(V\) be the variables in cell \((i, i + j)\).
   
   (b) Let \(U\) be the variables in cell \((i + j + 1, i + k)\).
   
   (c) For every pair \((A, B) \in V \times U\), if there is a rule of the form \(C \rightarrow AB\) in \(G\), then add \(C\) to cell \((i, i + k)\).

4. Accept if the start state is a variable in cell \((1, n)\).
```

Step 3 of the algorithm fills cells at increasing distances from the diagonal. For example, suppose the current cell is \((1, 3)\), which corresponds to \(w_1w_2w_3\). There are two ways to break this into nonempty substrings: \(w_1w_2w_3\) or \(w_1w_2 \circ w_3\). For each possible way to break it up, the cell corresponding to each substring is already filled in. If we let \(w_1w_2w_3 = x \circ y\), then \(V\) is the contents of the cell corresponding to \(x\) and \(U\) is the contents corresponding to \(y\). For each pair \((A, B) \in V \times U\), check if \(C \rightarrow AB\) is a rule in \(G\). If it is, add \(C\) to the cell \((1, 3)\) corresponding to \(w_1w_2w_3\).