CSCI 341–Fall 2024: Lecture Notes
Set 17: Complexity

Edward Talmage

December 2, 2024

Having looked at the broad-scale picture of what problems can and cannot be solved, we will now zoom back in a bit to consider only decidable languages. We know that these can all be decided, positively or negatively, in finite time. We would like to know something about how large those finite times may be.

Definition 1. Let M be a terminating algorithm (decider). The *time complexity* of M is given by a function $f : \mathbb{N} \to \mathbb{N}$ such that M halts in at most $f(n)$ computation steps on every input of length n.

We usually care about the asymptotic behavior of $f(n)$, so will express time complexity using big/little $O/\Omega/\Theta$ notation.

Example: Consider the decider we gave for $\{0^k1^k \mid k \geq 0\}$:

On input w:

1. Scan left to right to ensure input is in 0^*1^*, reset head left.
2. While there are 0's and 1's on the tape, scan across marking one 0 and one 1. Reject if cannot.
3. If everything is marked, accept. Else, reject

Step 1 takes $2n$ steps to walk across the input and back. Step 2 will repeat at most $n/2$ times, each iteration requires $O(n)$ time. Final scan to check is n steps. Total time is $O(n^2)$.

Definition 2. Let $t : \mathbb{N} \to \mathbb{R}^{\geq 0}$. The complexity class $Time(t(n))$ is the set of all decision problems that can be decided in $O(t(n))$ time.

- E.g. $\{0^k1^k \mid k \geq 0\} \in Time(n^2)$.

Exercise: Can you solve this problem faster?

On input w:

1. Check if $w \in 0^*1^*$, reset to left end.
2. As long as there are 0's or 1's on the tape:
   (a) Check parity of number of symbols on tape. Reject if odd.
   (b) Scan left to right, crossing off every other 0, every other 1.
3. If everything is crossed off, accept. Else, reject

Steps 1 and 3 are as before, but step 2 now only repeats $O(\log n)$ times, so our total runtime is $O(n \log n)$.

Exercise: Can you give a 2-tape TM that decides $\{0^k1^k \mid k \geq 0\}$ in $o(n \log n)$ steps?

On input w:

1. Ensure $w \in 0^*1^*$. Reset to left end of tape 1.
2. Scan left to right, copying all 0's to tape 2.

3. Reset tape 1 to first 1, tape 2 to left end.
4. Scan left to right on both tapes, crossing off 1's on tape 1 and 0's on tape 2. If one tape runs out of symbols before the other, reject.
5. If both tapes run out of symbols in the same step, accept.

This takes only $O(n)$ steps, since we traverse the input a constant number of times, regardless of its length. But this is not the same as having an $O(n)$ single tape TM. We know there is an equivalent single-tape TM, but the construction needed extra steps, and we are now counting steps.

Claim 1. *Let M be a k-tape decider with complexity $O(t(n))$, where $t(n) \geq n$. Then we can simulate M using a single-tape TM with complexity $O(t(n)^2)$.*

Proof. Recall that we can simulate a k tape machine M with a single-tape machine S by simulating the tapes one after another, separated by a divider symbol. For S to simulate one step of M, we need to look at the whole tape once to figure out what to do (collecting the symbols at the Read/Write heads of each tape), then once more to update the k Read/Write heads and tapes.

To compute the total time for this, we need to know how long the single-tape machine's tape is. At worst, each of M's k tapes can have length $t(n)$, if the head moves right in every step. Thus, S could need a tape of length $kt(n)$, and traverse it twice each step. Thus, the total work is at most $t(n)$ simulated steps times $2kt(n)$ work per step, for a total of $O(t(n)^2)$.

Aside: We restrict $t(n) \geq n$ so that we have time to set up the divider on the tape in S.

Aside: This is also a way to see why a TM with infinitely many tapes is nonsensical. We could technically simulate one, but it would have infinite time complexity.

□

This result covers most of the modifications we have considered for Turing Machines as a doubly-infinite tape is fundamentally similar to having two tapes, and this result applies to any finite number of tapes. It is actually quite notable that the complexity of having more tapes does not appear in the exponent of the complexity of our multi-tape simulator. Technically, the cost of simulating a k-tape machine is linear in k, but that is far better than being something else we could imagine, such as $t(n)^k$. Unfortunately, our simulation of non-deterministic Turing Machines is much more expensive.

Claim 2. *Let N be a non-deterministic Turing Machine of complexity $t(n) \geq n$. Then a (deterministic) TM M can simulate N with complexity $2^{O(t(n))}$.*

Proof Sketch: We discussed how to simulate an NTM using Breadth-First Search in the space of possible configurations, using an address tape to iterate through all configurations. Let b be the maximum branching factor of N's computation tree (the maximum number of options N non-deterministically chooses among). Since every path has length at most $t(n)$, the number of leaves in the computation tree is $O(b^{t(n)})$. The time to simulate each node is $O(t(n))$, so our total cost is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. We still need to convert our 3-tape simulation to a single-tape TM, but that at most squares the complexity, yielding $(2^{O(t(n))})^2 = 2^{2O(t(n))} = 2^{O(t(n))}$.

Aside: To see the reduction from $O(t(n)b^{t(n)})$ to $2^{O(t(n))}$:

$$\begin{aligned} T = O\left(t(n)b^{t(n)}\right) &= O\left(t(n)(2^{\log_2 b})^{t(n)}\right) \\ &= O\left(t(n)2^{dt(n)}\right) && \text{Where } d = \log_2 b \\ &\Rightarrow T \leq ct(n)^{dt(n)} && \text{for all } n \geq n_0, \text{ for some sufficiently large } n_0 \end{aligned}$$

Assume WLOG n_0 is large enough that $t(n) \geq 1$ for all $n \geq n_0$.

$$ct(n)2^{dt(n)} \leq 2^{ct(n)}2^{dt(n)} = 2^{(c+d)t(n)} = 2^{O(t(n))}$$

Example: Consider the Knapsack problem:

$$K = \left\{ \langle V, W, v_{lim}, w_{lim} \rangle \mid V,\ W \text{ are arrays of } n \text{ values, } \exists S \subseteq \{0, \ldots, n-1\} \text{ s.t. } \sum_{i \in S} w[i] \leq w_{lim} \text{ and } \sum_{i \in S} v[i] \geq v_{lim} \right\}$$

Exercise: Give an algorithm for deciding this problem on a non-deterministic TM.

On input $\langle V, W, v_{lim}, w_{lim} \rangle$:

1. Verify format
2. Non-deterministically "guess" a subset S of $\{0, \ldots, n-1\}$
3. Let $x = \sum_{i \in S} w[i]$, $y = \sum_{i \in S} v[i]$
4. Decide $(x \leq w_{lim})\&\&(y \geq v_{lim})$

Exercise: How can we non-deterministically "guess" every possible subset?

Each "guess" is just a different non-deterministic branch, and we will accept if any of the branches, and hence any of the subsets, accept. The trick is that we cannot have a 2^n-way non-deterministic branch, since a TM's state machine cannot change for different inputs. Instead, we need to use fixed branching to achieve the same end. To do this, consider the first item. Non-deterministically split into two branches, one which keeps that element and one which discards it. In each of those branches, consider the second element and split into a branch which keeps it and a branch which discards it. Continue in this way, splitting each of the previous branches into two for each consecutive element. After considering all n elements, we will have 2^n non-deterministic branches, one for each possible subset. This takes only $O(n)$ time, since each branch merely considered each element in order. Of course, that is $O(n)$ non-deterministic time, so we have a deterministic solution in $2^{O(n)}$ time.

Converting Optimization Problems to Decision Problems

How did we get the Knapsack Problem, an optimization problem, to be a decision problem? Typically, we ask what is the maximum value we can carry without exceeding the weight limit, not whether there is a subset of a particular value.

First, and most simply, if we can solve the optimization version of the problem, then we can solve the decision version, by finding the maximum possible value and comparing to the value limit. Second, if we have a solution for the decision version of the problem, we can solve the optimization version. To do this, realize that there are finitely many possible value limits, since the highest possible value would be the sum of the values of all the items (if the weight constraint were above the total weight of all items). We can then do a search between 0 and the sum of all values for the largest value which is actually achievable with a weight-valid set of items. This takes $O((\sum V)2^{O(n)})$ total time.

Since the values are not necessarily bounded in terms of n, this is not a great bound. Instead, we could just set the weight of every possible subset as v_{lim}, one after another, and see which valid subsets passed the highest v_{lim}. There are 2^n subsets, so we get a total time of $2^n * 2^{O(n)} = 2^{O(n)}$. Thus, we can solve either version of the problem with the other, without an asymptotic performance penalty.

What is most interesting here is actually the simpler reduction. Since we know that if we can solve the optimization problem efficiently, we can solve the decision problem equally efficiently, this tells us that if we cannot solve the decision problem efficiently, then neither can we solve the decision problem efficiently. This allows us to use decision problems to lower-bound the runtime of general computing problems, which is quite useful.