# Lecture 14: Rice's Theorem
# Introduction to Algorithmic Complexity

Ryan Bernstein

## 1 Introductory Remarks

- Assignment 3 is due today.
- Assignment 4 hasn't been posted yet because I'm really lazy

## 2 Examples of Mapping Reductions

**A Strange Example** Show that the problem of determining whether or not a Turing machine ever writes a blank character over a non-blank character is undecidable.

Even though this doesn't look like the problems we've been seeing, we can still show that $A_{TM}$ is mapping-reducible to this problem and therefore show that this problem is undecidable. To do this, we need to show a way to turn a candidate solution for $A_{TM}$ into a candidate solution for this problem. More specifically, we need to create a machine that will write a blank character over a non-blank character if and only if $M$ accepts $w$.

$F =$ "On input $\langle M, w \rangle$:

1. Modify $M$ to create a new machine $M'$ as follows:
   - Add a new character $c$ to $\Sigma$
   - On any transition that attempts to write a blank character over a non-blank character, write a $c$ instead
   - On any transition that attempts to read a blank character, add another transition to and from the same states that reads $c$ instead
   - On any transition to $q_{accept}$, first write a $c$ and then write a blank over it
2. Output $M'$"

**Worksheet Exercise** Let $L = \{\langle M \rangle \mid 1^* \subset L(M)\}$. Use a mapping reduction to show that $L$ is undecidable.

# 3 Rice's Theorem

On Tuesday, I introduced Rice's Theorem, which states that:

> Let $P$ be a non-trivial property of languages. Then $L(P) = \{\langle M \rangle \mid L(M) \text{ satisfies } P\}$ is undecidable.

This allows us to forgo mapping reductions in many cases, but we do have a couple of important restrictions here.

The first is that we require that $L(P)$ concerns only languages, not the machines themselves. $\text{E}_{\text{TM}} = \{\langle M \rangle \mid L(M) = \emptyset\}$ is an example of a predicate concerning the language of the machine, since the only thing that determines membership in $\text{E}_{\text{TM}}$ is a property of $L(M)$.

$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ halts on } w\}$, on the other hand, is based on a property of the *machine* — namely, whether or not it halts. Since looping and rejecting on $w$ both indicate that $w \notin L(M)$, the following two Turing machines both recognize the same language (namely $\{w \in \{0,1\}^* \mid w \text{ends in } 1\}$):

$M_1$ = "On input $w$:
1. If the last non-blank character in the input is a 0, REJECT
2. ACCEPT

$M_2$ = "On input $w$:
1. If the last non-blank character in the input is a 0, loop
2. ACCEPT

$\langle M_1, s \rangle \in \text{HALT}_{\text{TM}}$ for any $s$, but $\langle M_2, s \rangle \notin \text{HALT}_{\text{TM}}$, even though $L(M_1) = L(M_2)$. $\text{HALT}_{\text{TM}}$ is therefore based on the machine, rather than the language thereof.

Our other restriction is that our predicate must be *non-trivial*. We say that a predicate $P$ is non-trivial if the following conditions are met:

1. $L(P) \neq \emptyset$
2. $\overline{L(P)} \neq \emptyset$

In other words, this predicate must apply to some languages, but not *all* Turing machines. This means that a predicate like "$L(M)$ contains no duplicates" can't be shown to be undecidable via Rice's Theorem. In fact, it should be rather obvious that the language of a *trivial* predicate is decidable: we simply ACCEPT in every case or REJECT in every case.

## 3.1 Our Last Mapping Reduction(s): A Proof of Rice's Theorem

As our final in-class examples of mapping reductions, we'll create a general reduction with which we can (informally) prove Rice's Theorem. We're concerned only with non-trivial predicates about languages, which we can partition into two cases.

### 3.1.1 Case 1: $\emptyset \notin P$

Assume that $\emptyset$ does not satisfy $P$. Find one other Turing machine $N$ such that $L(N)$ *does* satisfy $P$. We can now show that $\text{A}_{\text{TM}} \leq_m L(P)$. To do so, we'll take a candidate solution for $\text{A}_{\text{TM}}$ (i.e. an encoding of a machine and a word) and transform it into a candidate solution for $L(p)$ (i.e. an encoding of a machine) so that the result is in $L(P)$ if and only if $M$ accepts $w$.

Since we know that $N$ satisfies $p$ based only on its language, we can create our new machine $M'$ such that $L(M') = L(N)$ if $M$ accepts $w$ and $L(M') = \emptyset$ otherwise.

$F$ = "On input $w$:

1. Construct a new machine $M'$ as follows: $M'$ = "On input $x$:
   (a) Simulate $M$ on $w$
      i. If $M$ accepts $w$, simulate $N$ on $x$
         A. If $N$ accepts $x$, ACCEPT $x$
         B. If $N$ rejects $x$, REJECT $x$
      ii. If $M$ rejects $w$, REJECT $x$"
2. Output $\langle M' \rangle$"

#### 3.1.2 Case 2: $\emptyset \in L(P)$

Since we're showing that $L(P)$ is undecidable using an indirect proof, we start with the assumption that $L(P)$ is decidable. This means that $L(\neg P)$ is also decidable. We can therefore use the same proof as in case 1, showing that $L(\neg P)$ is undecidable, which implies that $L(P)$ is undecidable as well.

## 4 Introduction to Complexity Analysis

Until this point, all we've concerned ourselves with in this class has been computability. All we've been doing is showing what it was and was not possible to compute using various types of machines. We haven't cared at all about efficiency.

Efficiency is important, though. Based on the scale of the programs that we write for school, we often think of efficiency in terms of whether or not we have time to make a cup of coffee before our program finishes compiling or running, but as the size of the problems we face increases, so too does the importance of *efficient* algorithms to solve some problem, rather than just algorithms.

Consider this: RSA encryption uses keys that are 1024 or 2048 bits in length. Let's say that you're security conscious, so you encrypt your hard drive (or your email, or your network communications) using a 2048-bit key.

It's pretty trivial to write an algorithm that can crack your encryption key (or in fact, *any* encryption key). We just try them all in sequence, starting with $0^{2048}$, then $0^{2047}1$, then $0^{2046}10$, and so on. If I can test 10,000 keys per second, it will take me $10^{606}$ years to exhaust the possible key space. Since I plan to be dead sometime in the next $10^2$ years, this algorithm, while completely correct, is not incredibly useful to me.

Whether or not a problem is computable is a useful question to ask ourselves, to be sure. But perhaps equally important is the question of whether or not that problem can be solved *in a reasonable amount of time*. That is the question to which we now turn our attention.

This means that, for the remainder of the term, we will be talking about only problems that are Turing-decidable. All of these problems can be solved, but we'll now be dividing the class of decidable problems into subcategories based on *how quickly* they can be solved.

## 4.1 Asymptotic Notation

To measure the relative runtimes of various algorithms, we use what's called *asymptotic notation*. You've probably seen a brief introduction to asymptotic notation in classes like CS 163. The most common notation we use is "Big-O" notation, which provides an *upper bound* for the number of steps required by an algorithm as a function of the size of the input. Because this is an upper bound, Big-O notation is sometimes referred to as "worst-case" complexity.

This requires a definition of two things:

1. What is a step?
2. What is the size of the input?

For a Turing machine, these are both fairly simple concepts. Each step is a single transition followed in the machine. The length of the input is simply the length of the string passed to the machine.

### 4.1.1 Definition of Big-O

Let $f$ and $g$ be functions from $\mathbb{N}$ to $\mathbb{R}^+$. $f(n) = O(g(n))$ if positive integers $c$ and $n_0$ exist such that for every integer $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

This means that $n^2 = O(n^3)$, since $n^3$ should be greater than $n^2$ for all positive values of $n$. But it also means that $n^2 = O(\frac{1}{2}n^2)$, even though $n^2 \geq \frac{1}{2}n^2$ for all $n \geq 0$. This is because of the constant factor, $c$.

We can still say that $n^2 \leq c \cdot \frac{1}{2}n^2$ for all $n \geq 0$ and $c \geq 2$. Therefore, $n^2 = O(\frac{1}{2}n^2)$, and $\frac{1}{2}n^2 = O(n^2)$.

### 4.1.2 Simplification

This means that we disregard all constant factors when analyzing runtime complexity. It also means that if $f$ is some polynomial $a_k n^k + a_{k-1}n^{k-1} + ... + a_1 n + a_0$, we can disregard all but the highest-order term and say that $f(n) = O(n^k)$.

What terms do we keep? Here is an incomplete list of common running times in no particular order.

- Polynomial running times (e.g. $O(n^2)$) like those just discussed
- *Linear* running times, written $O(n)$
- *Constant* runnning times, written $O(1)$. These take the same amount of time regardless of the size of the input. These algorithms are ideal.
- *Logarithmic* running times, written $O(\log n)$. While simplifying, we disregard log bases, since these can all be shown to be equivalent using constants and log rules. I never remember the log rules, so I'm happy not worrying about it.
- Running times such as $O(n \log n)$, which are sometimes referred to informally as "linearithmic"

- *Factorial* running times, written $O(n!)$. These dwarf any polynomial
- *Exponential* running times, written $O(2^n)$. These are always written with a base of two, again because of weird log rules. Exponential running times are also very bad.

We disregard quite a lot of information when analyzing running times, which means that for many problems, asymptotic analysis doesn't provide a good representation of how much actual wall time it takes to compute something. For instance, drawing an $n \times n$ frame of a single solid color in a computer's video buffer could be said to take $n^2$ steps (times the computation time for an individual pixel), since we need to compute and write the value for every pixel on the screen.

Doing this on a GPU can make it run much faster, since it parallelizes the process. But since we're *still* drawing every pixel, the runtime remains $n^2$. Asymptotic analysis doesn't take parallelism into account at all.

The $n_0$ at which one function overtakes another may also be very large.

Why do we do use asymptotic analysis, then?

- Because we're concerned with the way that the running times of these algorithms grow as we add more and more input.
- Because when we get into differences of $n^2$ versus $n!$ or $2^n$, we can hit situations like those discussed in the encryption example, where solving a problem can take so long as to make the algorithm useless. Trying every possible 2048-bit encryption key, for example, is $O(2^n)$.

## 4.2 Time Complexity Classes

This definition of asymptotic notation allows us to define an infinite number of time complexity classes. We say that $\textsc{Time}(t(n)$ is the set of all languages decidable by an $O(t(n))$ time Turing machine. Note that time complexity classes are sets of languages, not sets of Turing machines. In other words, just because a Turing machine that runs in a given amount of time exists to decide some language, this does not mea that the language *isn't* a member of a faster complexity class; we may just have yet to find a Turing machine to decide the same problem that runs faster.

### 4.2.1 In Practice

**Example** Consider the following Turing machine, which decides $\{0^n1^n \mid n \geq 0\}$:

$M =$ "On input $w$:

1. If $w$ is not of the form $0^*1^*$, Reject
2. While uncrossed zeros *and* ones remain:
   (a) Scan across the tape from left to right. Cross off the first zero and the first one
   (b) Return to the left end of the tape
3. Scan the input. If uncrossed zeros *or* uncrossed ones remain, Reject
4. Accept"

To find the running time of the machine, we can simply sum the running time of each individual step.

Line 1: Ensuring that $w$ is of the form $0^*1^*$

This requires a pass across the entire input, ensuring that a zero never follows a one. Therefore, this step is $O(n)$

Line 2: The loop

To analyze loops, we find the time complexity of the loop body and then multiply it by the number of times that the loop executes.

Since the loop should cross off two characters each time, it will execute $\frac{n}{2}$ times.

Line 2a: Crossing off a zero and a one

In the worst case, the one that we cross off is the last character in the input. We therefore say that the worst-case time-complexity of this step is $O(n)$.

Line 2b: Returning to the left side of the input

In the worst case, this requires us to travel from the end of the string back to the beginning. We therefore say that the worst-case complexity of this step is $O(n)$.

The body of our loop is therefore $O(2n)$. Multiplying this by the $\frac{n}{2}$ times that the loop executes therefore yields $O(n^2)$.

Line 3: Scanning for uncrossed characters

This requires another full pass across the input, and is therefore $O(n)$

Line 4: Accept

This takes the same number of steps regardless of the size of the input. We therefore say that this is $O(1)$

The total running time of our algorithm is therefore:

$$O^{n^2} + 2O^n + O(1)$$

We simplify this to $O(n^2)$.

**Example** Find the runtime of the following Turing machine, which decides $\{\langle G\rangle \mid G \text{ is a connected undirected graph}\}$

$M =$ "On input $\langle G\rangle$:

1. Mark a node
2. Repeat until no new nodes are marked:
   (a) Mark any node reachable from an already-marked node
3. Iterate over $V$. If any unmarked nodes remain, Reject
4. Accept

Line 1: Mark a node

This is pretty quick. Since we just mark the first one, this is $O(1)$

Line 2: The loop

At worst, this loop will repeat until every node in $V$ is marked, so this loop executes $|V|$ times.

Line 2a: Mark any node reachable from an already-marked node

This requires us to iterate over 1) all of the marked nodes and 2) all of the edges in the edge set to find which nodes are reachable from marked nodes. We can therefore say that this line takes $O(|V| \cdot |E|)$ steps in the worst case.

This entire loop then has runtime $O(|V| \cdot |V| \cdot |E|) = O(|V|^2 \cdot |E|)$

Line 3: Check for unmarked nodes

This requires us to iterate over $V$, so this line has runtime $O(|V|)$

Line 4: ACCEPT

This is a constant-time operation ($O(1)$).

We therefore say that this machine runs in $O(|V|^2 \cdot |E|)$.

# 5 Variants of Turing Machines

We went through quite a bit of work to introduce Turing machine variants, and then decided that they were equivalent in power to regular Turing machines. While using them can make deciding certain problems easier, we haven't actually seen a lot of examples where this was the case.

One place where multi-tape and non-deterministic machines *do* shine is in time complexity analysis. While a standard Turing machine and a multi-tape machine are capable of deciding the same class of problems, it can also be the case that a multi-tape Turing machine can do it much faster.

**Worksheet Exercise** Consider the following two-tape Turing machine that also decides $\{0^n1^n \mid n \geq 0\}$:

$M$ = "On input $w$:

1. If the input is not of the form $0^*1^*$, REJECT
2. For each zero in the input, write a one to tape 2
3. Return tape 2's head to the left
4. For each one in the input, ensure that we see a one on tape 2
   - If either tape sees a blank before the other, REJECT
   - If both tapes read blanks at the same time, ACCEPT"

Find the time complexity of $M$.

**Theorem** Every $t(n)$-time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

## 5.1 Runtime in Nondeterministic Turing Machines

Nondeterministic machines allow us to "branch" our computation, creating a tree. We define the running time of a nondeterministic machine as the maximal number of steps on any individual branch in the computation tree (i.e. the depth of the tree).

**Theorem** Every $t(n)$-time nondeterministic decider has an equivalent $2^{O(t(n))}$ time deterministic Turing machine.

I really don't feel like proving these things right now. The proofs come from the simulations of multitape and non-deterministic machines that we saw in class. The proofs are also in the book in section 7.1.