Computational Lower Bounds Using Diagonalization

1. Languages, Turing Machines and Complexity Classes

M V Panduranga Rao

M V Panduranga Rao works at the Tata Research Development and Design Centre, Pune, India.
His interests lie mainly in theoretical computer science.

What can a computer with limited resources like time and space accomplish? Can it solve our favourite computational problem? These are the kind of questions that we implicitly ask when designing 'efficient algorithms'. It is also interesting to know which problems cannot be solved with computers operating with limited resources, no matter how smart we are as algorithm designers. Moreover, given a problem, we would like to know the lower bound on such resources required to solve it using a given computer. This article in two parts, discusses an important technique called diagonalization for establishing such lower bounds. In this part we will fix a model of a computer – indeed, one that is as powerful as any other known mechanical model – and explore some important features of this model. In the second part, we will introduce diagonalization, its applications and potential shortcomings.

Keywords
Languages, decision problems, Turing machines, Universal Turing machine, complexity classes.

1. Languages, Problems and Computation

Consider the set $\{0,1\}$. Call this set an *alphabet* and denote it by Σ. A *word* or a *string* is a concatenation of symbols drawn from Σ. We call the set of all words formed from such concatenations Σ^*. Thus, $\Sigma^* = \{0,1,00,01,10,11,\ldots\}$. A language L is a subset of Σ^*. Notationally, $L \subseteq \Sigma^*$. Further, we denote the length of a string w by $|w|$.

The first obvious problem that one can think of in this setting is: Given a word $w \in \Sigma^*$, does it belong to a given language L? This may not necessarily be an easy question to answer. More so if L is an infinite

set. Further, one problem might be 'easier' than the other. Somehow, deciding whether a given integer belongs to the set of even integers seems easier than deciding whether it belongs to the set of primes. This article concerns such *decision problems* and the difficulty of solving them. In some places, for the purpose of exposition, we use the words "yes" and "no" for the answer to a decision problem. This does not go beyond the alphabet because we could use (say) 1 for "yes" and 0 for "no" just as well. Indeed, we will use them interchangeably.

Somehow, deciding whether a given integer belongs to the set of even integers seems easier than deciding whether it belongs to the set of primes.

2. Models of Computation

An algorithm is a step-by-step listing of instructions executed to carry out a (computational) task. Consider the problem of deciding whether a string $x \in \Sigma^*$ is a prime number or not. A naive algorithm that immediately springs to mind is:

1. *First* set a counter i to 10 (that is, 2 in binary).
2. Is x divisible by i?
3. *if* yes, output "no" (x is not prime),
4. *else* increment i by 1.
5. Is i equal to $\lfloor\sqrt{x}\rfloor$?
6. *if* yes, output "yes" (x is prime).
7. *else goto* step 2.

One can run through these steps on a piece of paper. But suppose we wish to get this done 'mechanically'. That is to say, we seek a machine that accomplishes the task for us. That way, we can also hope to have a well-defined way of measuring the difficulty of a problem. The number of steps for which the machine runs, and the space it takes up for solving the problem will allow

The power of this model is apparent by what is called the Church–Turing thesis: the Turing machine is capable of performing any computation that can be performed 'mechanically'!

us to measure the goodness of the algorithm. Lower-bounds on these resources would indicate the difficulty of the problem. That said, we would not like to bestow unreasonable power to our mechanical computer. We therefore insist that at any step, it should do only a finite amount of work.

Consider what are required. First, we need some space to store the input x and some scratch space for doing the division. Second, we need a way of capturing the logic of the algorithm: statements like *first*, *if*, *then* and *else*.

It turns out that a mechanical model called the Turing machine fits the bill exactly. A Turing machine is an abstract computer that captures the notion of an algorithm nicely. It was proposed in 1936 by Alan Turing, a great British mathematician and logician. The power of this model is apparent by what is called the Church–Turing thesis: the Turing machine is capable of performing any computation that can be performed 'mechanically'!

[1] Variants of Turing machines like those having multiple tapes and tapes that extend to infinity in both directions exist. But it can be shown that all are equivalent to the single-tape version in what can be ccomputed.

3. Turing Machines

A Turing machine basically consists of two components. The first is a tape (that extends to infinity in one direction)[1] divided into cells, each of which can hold either a symbol from Σ or one of the *tape symbols*: the blank $\sqcup$ or an end marker \$. The tape is also endowed with a *head* for reading and writing symbols into the tape cells one at a time. This tape is used for storing the input, the output and all the scratch-space required by the algorithm[2]. For example, the input x, the space required to store i, for carrying out division of x by i and for calculating $\lfloor\sqrt{x}\rfloor$ can be accommodated on the tape. The second component is a finite control that captures the logic of the algorithm. Just as a human brain goes through different stages when solving the problem, so does the Turing machine. For instance, there is an

[2] In our case, the output is simply a "yes" or "no" answer to the decision problem. Such a Turing machine M, also called a decider, is a function M:{0,1}* $\rightarrow$ {yes,no, $\uparrow$}. Here, $\uparrow$ denotes the situation when M does not halt on the input: it enters an infinite loop. In general, Turing machines can output strings: for instance, the sine of an angle up to so many decimals.

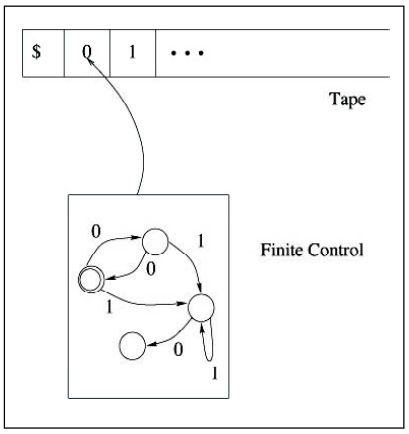

Figure 1. The single-tape Turing machine.

initial stage when no number has been tried yet and the counter i has to be initialized. Then there is a stage when different i's are being tried out, and finally a stage when some i has succeeded in dividing x or all of the numbers till $\lfloor\sqrt{x}\rfloor$ have failed in dividing x perfectly. The Turing machine is equipped with a finite set S of states corresponding to the various stages of the algorithm. Naturally, in this set of states lies a special start state s_0 and a subset F of *accepting* or *final* states. In our case, we enter into a state in F if and only if x is a prime, and accept by outputting "yes".

With these components in hand, we can define precisely how the Turing machine transits from one stage to another – in effect, executes the algorithm. A transition function $\delta : S \times \{\Sigma \cup \{\$, \sqcup\}\} \rightarrow S \times \{\Sigma \cup \{\$, \sqcup\}\} \times \{-1, 0, +1\}$ does the job for us.

Let us see how the machine works. If the machine is in state s, and the tape head is reading σ, let $\delta(s, \sigma) = (s', \sigma', +1)$. Then, the transition function writes the symbol σ' into the cell, moves the head to the right and the machine enters the state s'. Given this definition, note that the transition function can be spelt out as a finite set of ordered tuples of the form $(s, \sigma, s', \sigma', d)$, where $d \in \{-1, 0, +1\}$. The state set and the transition function therefore form the meat of an algorithm. The

The state set and the transition function form the meat of an algorithm.

The quintuple $(S, \Sigma, s_0, \delta, F)$ completely defines a Turing machine. We say that a machine M decides a language L if, given an input $x \in \Sigma^*$, it outputs "yes" (or, $M(x) = 1$) if $x \in L$ and "no" ($M(x) = 0$) otherwise, in a finite amount of time. We denote by $L(M)$ the language decided by a machine M.

Observe that the contents of the tape, the current state and the position of the tape head define the *instantaneous configuration* of a Turing machine completely.

4. Encoding Turing Machines as Strings

A useful fact is that Turing machines can be encoded as strings. This will allow us to pass a Turing machine as input to another Turing machine. In fact, we will even be able to pass a Turing machine as input to itself. Let $M = (S, \Sigma, s_0, \delta, F)$ be a Turing machine. Let the string defining M begin with the encoding of the set of states. We encode the set of states $S = \{s_0, s_1, \ldots, s_k\}$ as $01001\ldots\underbrace{0\ldots0}_{k}11$, where the 1's act as separators. The last two 1's indicate that the list of states has ended. We now encode symbols: $|\Sigma|$ of the alphabet, the tape symbols, and the three symbols for head movement. These are again represented in unary. Thus, the ith of these $|\Sigma| + 5$ symbols is represented by $|S| + i$ 0's, separated from one another by 1's. The list of symbols again ends with two 1's. Now we encode the ordered tuples $(s, \sigma, s', \sigma', d)$ of the transition function. We already have the unary strings for s, σ, s', σ' and d. We separate the elements of a tuple by a 1 and the tuples themselves by two 1's. Thus, the tuple $(s, \sigma, s', \sigma', d)$ will be $11\underbrace{0\ldots0}_{\langle s\rangle}1\underbrace{0\ldots0}_{\sigma}1\underbrace{0\ldots0}_{\langle s'\rangle}1\underbrace{0\ldots0}_{\sigma'}1\underbrace{0\ldots0}_{d}11$. This gives us an encoding of the transition function. The set of tuples is ended by three 1's after which follows the encoding of the set of final states.

A useful fact is that Turing machines can be encoded as strings. This will allow us to pass a Turing machine as input to another Turing machine.

Once we fix this encoding, every string in Σ^* describes a

unique Turing machine. We will write the encoding of a Turing machine M as $\langle M \rangle$. A little thought will reveal that infinitely many encoding schemes are possible.

5. Universal Turing Machines

The Turing machines that we have described are problem specific. Each Turing machine essentially corresponds to an algorithm meant to solve a specific problem. A Turing machine that tells if a given number is even or not cannot tell if it is prime or not.

Wouldn't it be nice to have a Universal Turing machine U that can simulate any other Turing machine M? By simulation, we mean that U must output the same answer as M on the same input. Fortunately, the ability to encode M as a string makes this possible. As Alan Turing discovered, it is possible to construct such a Universal Turing machine U. Without going into details, we remark that the Universal Turing machine may be seen as a three-tape Turing machine. On one tape lies the description $\langle M \rangle$ of the Turing machine M to be simulated. On the second tape lies the input x on which to simulate M. The transition function of U is such that it reads $\langle M \rangle$, interprets the transition function of M from $\langle M \rangle$ and executes the same steps on x as M would have done, using the third tape. A celebrated result of Alan Turing says something more: *there exists a Universal Turing machine that can simulate T steps of any given Turing machine M in at most $cT \log T$ steps, where c is independent of x and depends only on $|\langle M \rangle|$, the length of $\langle M \rangle$.*

Indeed, the general purpose computer that we are so familiar with today is essentially a Universal Turing machine, and the programs that we run on them are like encodings of specific Turing machines.

The general purpose computer that we are so familiar with today is essentially a Universal Turing machine, and the programs that we run on them are like encodings of specific Turing machines.

We may permit the transitions among the states to be governed, instead of a function, by a *relation*. In other

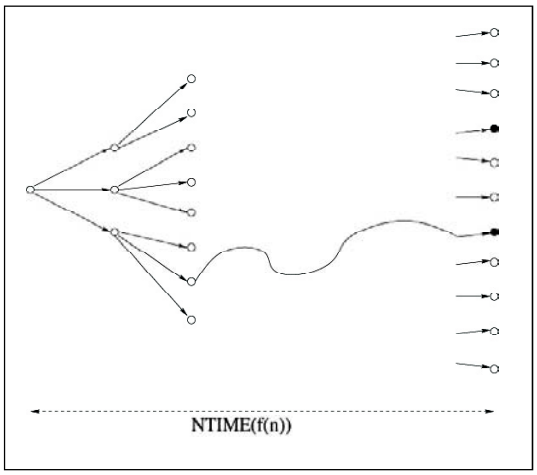

Figure 2. Nondeterministic computation tree. The shaded circles represent accepting configurations.

words, $\delta(s, \sigma)$ is now potentially a set $\{(s_1, \sigma_1, d_1), \ldots, (s_k \sigma_k, d_k)\}$, such that $s_i \in S$, $\sigma_i \in \Sigma$, and $d_i \in \{-1, 0, +1\}$, for $1 \leq i \leq k$. Thus, from the same instantaneous configuration, computation can *nondeterministically* branch out into several instantaneous configurations at the next step. Therefore computation on this variant is representable as a *tree*, with instantaneous configurations as nodes (see *Figure* 2). The depth of the tree gives the running time of the machine. As against the previous *deterministic* variant, such a Turing machine is called a *nondeterministic* Turing machine. A nondeterministic Turing machine is said to accept an input if *any* path in the computation tree ends in an accepting state.

A nondeterministic Turing machine is said to accept an input if *any* path in the computation tree ends in an accepting state.

6. Computational Complexity Classes

It is logical and reasonable to measure the time (or space) taken by an algorithm to solve an instance of a problem in terms of its input size. For example, if n is the size of the input, an algorithm can be said to take $f(n)$ time (or space) where $f : N \rightarrow N$. A technical requirement we will not bother about is that the function f should be time-constructible: a function f is time-constructible if there exists a Turing machine that, when given 1^n as input, outputs $1^{f(n)}$ in $O(f(n))$ time[3]. Most functions are time-constructible: n^2, $\log n$, 2^n, etc.

[3] Note that this Turing machine is not a decider; its output is a string.

We can also speak of a class of languages that can be decided in time $f(n)$ by deterministic Turing machines. We denote such a class by DTIME($f(n)$). Consider for example the language of palindromes: $L_{\text{pal}} = \{x \in \Sigma^* \mid x \text{ reads the same as its reverse } x^R\}$. A simple algorithm (read Turing machine) for deciding L_{pal} iteratively strikes off symbols from either end of the input if they match. If at any iteration, there is a mismatch, the algorithm rejects x. On the other hand, if one or no symbols remain as a result of the above process, then x is indeed a palindrome and is accepted. The number of steps that the above Turing machine would take on an input of length $|x| = n$ is at most $(n+2)(n+4)-1$. Thus, $L_{\text{pal}} \in DTIME((n+2)(n+4)-1)$. Observe that we worry about the number of steps in the worst case. While there are several inputs that require much less than $f(n)$ steps, there also exist inputs that would require those many. Similarly, the class of languages decided by nondeterministic Turing machines in time $f(n)$ is denoted by NTIME($f(n)$). We denote the class of languages decidable by a deterministic Turing machine in time polynomial in the length of the input by P. In other words, $P = \cup_{c\geq 1}\text{DTIME}(n^c)$. Similarly EXP (for exponential) denotes $\cup_{c\geq 1}\text{DTIME}(2^{n^c})$. The class of languages decidable by nondeterministic Turing machines in polynomial time is called the class NP: $NP = \cup_{c\geq 1}\text{NTIME}(n^c)$.

We denote the class of languages decidable by a deterministic Turing machine in time polynomial in the length of the input by *P*.

The resources (in terms of time, space, nondeterminism, etc.) made available to a given model of computation define various complexity classes. Of particular interest are robust classes: classes whose definition does not change between reasonable models of computation. P and NP are examples of robust classes.

The class of languages decidable by nondeterministic Turing machines in polynomial time is called the class *NP*.

The central objective of computational complexity theory is to decide whether two given classes are the same, or if one is properly contained in the other. The first entails showing that whatever languages can be decided

The central objective of computational complexity theory is to decide whether two given classes are the same, or if one is properly contained in the other.

using the resources corresponding to one class can also be decided using the resources of the other class. If the two classes are *separate*, there exists a language that can be (i) decided by a machine using the resources corresponding to one complexity class, but (ii) cannot be decided by *any* machine using the resources corresponding to the other class. In the second case, we show that deciding the language requires more than the resources that define the complexity class. In other words, we have found a *lower bound* on the resources needed to solve the problem on this model of computation.

For example, consider the classes P and NP. It is easy to see that $P \subseteq NP$; to simulate a polynomial time deterministic machine that recognizes a language in P by a polynomial time nondeterministic machine, we just ignore the extra 'resource' of nondeterminism. Showing that P is a proper subset of NP would involve showing a super-polynomial lower bound on the time required to decide some language in NP. Whether P is equal to NP or a proper subset is a central open question of complexity theory.

In the next part, we will study diagonalization, an important lower-bounding tool in a complexity theorist's kit.

Address for Correspondence
M V Panduranga Rao
Tata Research Development and Design Centre
Pune, India
Email: mvpandurangarao.m@tcs.com

Suggested Reading

[1] Sanjeev Arora and Boaz Barak, *Complexity Theory: A Modern Approach*, Web draft available at http://www.cs.princeton.edu/theory/complexity/

[2] J E Hopcroft, R Motwani and J D Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, 2006.