1. Many computational tasks involve searching for a solution across a vast space of possibilities (for example, the aforementioned tasks of solving linear equations and finding a maximal set of invitees to a dinner party). Is there an *efficient* search algorithm for all such tasks, or do some tasks inherently require an exhaustive search?

Since we cannot very well check every one of the infinitely many possible algorithms, the only way to verify that the current algorithm is the best is to *mathematically prove* that there is no better algorithm. This may indeed be possible to do, since computation can be given a mathematically precise model. There are several precedents for proving *impossibility results* in mathematics, such as the independence of Euclid's parallel postulate from the other basic axioms of geometry, or the impossibility of trisecting an arbitrary angle using a compass and straightedge. Such results count among the most interesting, fruitful, and surprising results in mathematics.

Given the above discussion, it is no surprise that mathematical proofs are the main tool of

1.1.3 Big-Oh notation

As mentioned above, we will typically measure the computational efficiency algorithm as the number of a basic operations it performs as *a function of its input length*. That is, the efficiency of an algorithm can be captured by a function T from the set of natural numbers $\mathbb{N}$ to itself such that $T(n)$ is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function is sometimes be overly dependant on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of single digit *binary* (i.e., base 2) numbers as a basic operation, as opposed to *decimal* (i.e., base 10) numbers. To help us ignore these low level details and focus on the big picture, the following well known notation is very useful:

DEFINITION 1.2 (BIG-OH NOTATION)
If f, g are two functions from $\mathbb{N}$ to $\mathbb{N}$, then we **(1)** say that $f = O(g)$ if there exists a constan
such that $f(n) \leq c \cdot g(n)$ for every sufficiently large n, **(2)** say that $f = \Omega(g)$ if $g = O(f)$, **(3)**
that $f = \Theta(g)$ is $f = O(g)$ and $g = O(f)$, **(4)** say that $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g$
for every sufficiently large n, and **(5)** say that $f = \omega(g)$ if $g = o(f)$.

To emphasize the input parameter, we often write $f(n) = O(g(n))$ instead of $f = O(g)$, a
use similar notation for $o, \Omega, \omega, \Theta$.

The notion of *computation* has existed in some form for thousands of years. In its everyday meaning, this term refers to the process of producing an output from a set of inputs in a finite number of steps. Here are three examples for computational tasks:

> *"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations."*
> Alan Turing, 1950

We start with an informal description of computation. Let f be a function that takes a string of bits (i.e., a member of the set $\{0,1\}^*$) and outputs, say, either 0 or 1. Informally speaking, an *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0,1\}^*$. The set of rules being followed is fixed (i.e., the same rules must work for all possible inputs) though each rule in this set may be applied arbitrarily many times. Each rule involves one or more of the following "elementary" operations:

1. Read a bit of the input.

2. Read a bit (or possibly a symbol from a slightly larger alphabet, say a digit in the set $\{0, \ldots, 9\}$) from the "scratch pad" or working space we allow the algorithm to use.

   Based on the values read,

3. Write a bit/symbol to the scratch pad.

4. Either stop and output 0 or 1, or choose a new rule from the set that will be applied next.

Formal definition. Formally, a TM M is described by a tuple (Γ, Q, δ) containing:

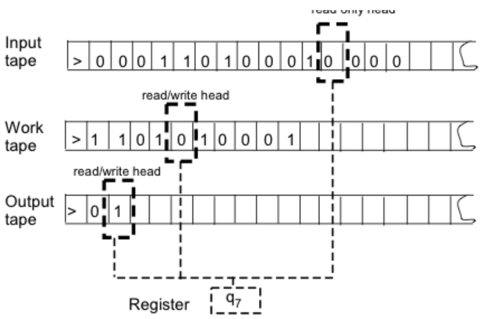
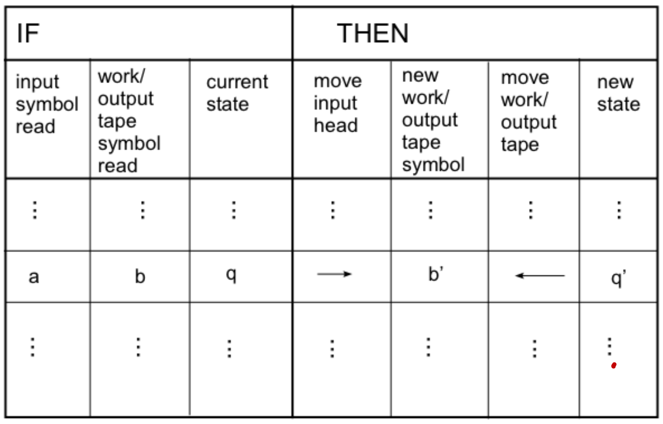


Figure 1.1: A snapshot of the execution of a 3-tape Turing machine M with an input tape, a work tape, and an output tape.

- A set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\square$, a designated "start" symbol, denoted $\triangleright$ and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted q_{start} and a designated halting state, denoted q_{halt}.
- A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$ describing the rule M uses in performing each step. This function is called the *transition function* of M (see Figure 1.2.)



IF			THEN			
input symbol read	work/ output tape symbol read	current state	move input head	new work/ output tape symbol	move work/ output tape	new state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	→	b'	←	q'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.2: The transition function of a two tape TM (i.e., a TM with one input tape and one work/output tape).

An important special case of functions mapping strings to strings is the case of *Boolean* functions, whose output is a single bit. We identify such a function f with the set $L_f = \{x : f(x) = 1\}$ and call such sets *languages* or *decision problems* (we use these terms interchangeably). We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language L_f (i.e., given x, decide whether $x \in L_f$).

Definition 1.4 (Computing a function and running time)
Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ and let $T : \mathbb{N} \rightarrow \mathbb{N}$ be some functions, and let M be a Turing machine. We say that M *computes* f *in* $T(n)$*-time*[2] if for every $x \in \{0,1\}^*$, if M is initialized to the start configuration on input x, then after at most $T(|x|)$ steps it halts with $f(x)$ written on its output tape.
We say that M *computes* f if it computes f in $T(n)$ time for some function $T : \mathbb{N} \rightarrow \mathbb{N}$.

Claim 1.8
For every $f : \{0,1\}^* \rightarrow \{0,1\}$ *and time-constructible* $T : \mathbb{N} \rightarrow \mathbb{N}$, *if* f *is computable in time* $T(n)$ *by a TM* M *using alphabet* Γ *then it is computable in time* $4 \log |\Gamma| T(n)$ *by a TM* $\tilde{M}$ *using the alphabet* $\{0, 1, \Box, \triangleright\}$.

1.5 Deterministic time and the class P.

A *complexity class* is a set of functions that can be computed within a given resource. We will now introduce our first complexity classes. For reasons of technical convenience, throughout most of this book we will pay special attention to Boolean functions (that have one bit output), also known as *decision problems* or *languages*. (Recall that we identify a Boolean function f with the language $L_f = \{x : f(x) = 1\}$.)

DEFINITION 1.19 (THE CLASS **DTIME**.)
Let $T : \mathbb{N} \rightarrow \mathbb{N}$ be some function. We let $\mathbf{DTIME}(T(n))$ be the set of all Boolean (one bit output) functions that are computable in $c \cdot T(n)$-time for some constant $c > 0$.

The following class will serve as our rough approximation for the class of decision problems that are efficiently solvable.

DEFINITION 1.20 (THE CLASS **P**)
$\mathbf{P} = \cup_{c\geq 1}\mathbf{DTIME}(n^c)$

As mentioned above, the complexity class **NP** will serve as our formal model for the class of problems having efficiently verifiable solutions: a decision problem / language is in **NP** if given an input x, we can easily verify that x is a YES instance of the problem (or equivalently, x is in the language) *if* we are given the polynomial-size *solution* for x, that *certifies* this fact. We will give several equivalent definitions for **NP**. The first one is as follows:

DEFINITION 2.1 (THE CLASS **NP**)
A language $L \subseteq \{0,1\}^*$ is in **NP** if there exists a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,u) = 1$$

If $x \in L$ and $u \in \{0,1\}^{p(|x|)}$ satisfy $M(x,u) = 1$ then we call u a *certificate*[1]for x (with respect to the language L and machine M).

Claim 2.3
$\mathbf{P} \subseteq \mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$.

Proof: ($\mathbf{P} \subseteq \mathbf{NP}$): Suppose $L \in \mathbf{P}$ is decided in polynomial-time by a TM N. Then $L \in \mathbf{NP}$ since we can take N as the machine M in Definition 2.1 and make $p(x)$ the zero polynomial (in other words, u is an empty string).

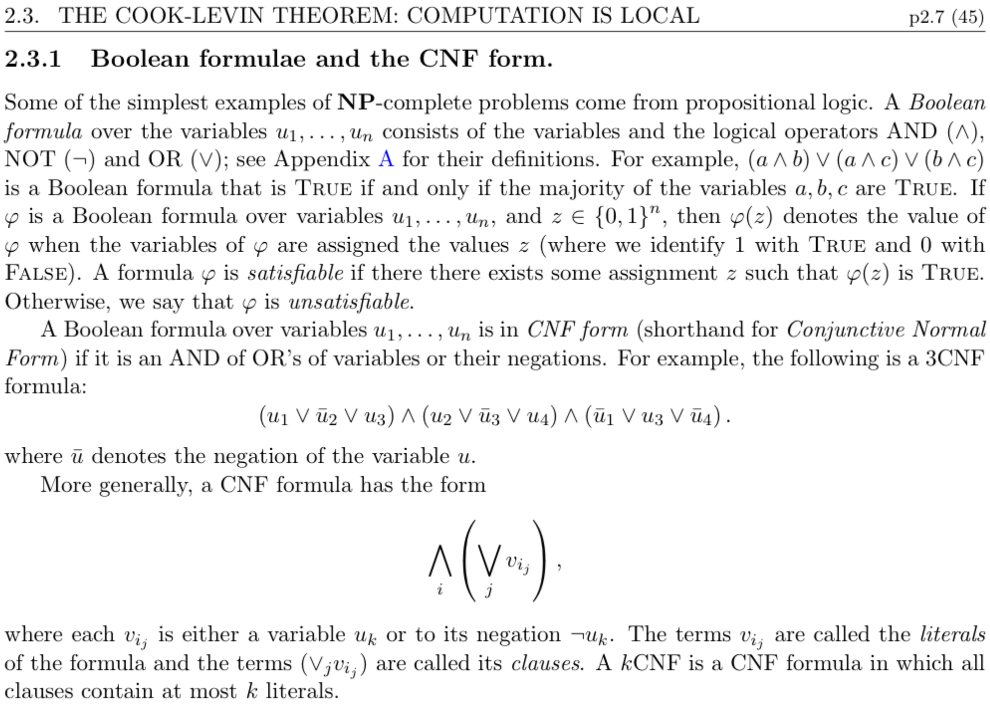
($\mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$): If $L \in \mathbf{NP}$ and $M, p()$ are as in Definition 2.1 then we can decide L in time $2^{O(p(n))}$ by enumerating all possible u and using M to check whether u is a valid certificate for the input x. The machine accepts iff such a u is ever found. Since $p(n) = O(n^c)$ for some $c > 1$ then this machine runs in $2^{O(n^c)}$ time. Thus the theorem is proven. ■

At the moment, we do not know of any stronger relation between $\mathbf{NP}$ and deterministic time classes than the trivial ones stated in Claim 2.3. The question whether or not $\mathbf{P} = \mathbf{NP}$ is considered *the* central open question of complexity theory, and is also an important question in mathematics and science at large (see Section 2.7). Most researchers believe that $\mathbf{P} \neq \mathbf{NP}$ since years of effort has failed to yield efficient algorithms for certain $\mathbf{NP}$ languages.

It turns out that the independent set problem is at least as hard as any other language in **NP**: if it has a polynomial-time algorithm then so do all the problems in **NP**. This fascinating property is called **NP**-*hardness*. Since most scientists conjecture that $\mathbf{NP} \neq \mathbf{P}$, the fact that a language is **NP**-hard can be viewed as evidence that it cannot be decided in polynomial time.

How can we prove that a language B is at least as hard as some other language A? The crucial tool we use is the notion of a *reduction* (see Figure 2.1):

Definition 2.7 (Reductions, **NP**-hardness and **NP**-completeness)
We say that a language $A \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible to a language* $B \subseteq \{0,1\}^*$ (sometimes shortened to just “polynomial-time reducible”[2]) denoted by $A \leq_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.
We say that B is **NP**-*hard* if $A \leq_p B$ for every $A \in \mathbf{NP}$. We say that B is **NP**-*complete* if B is **NP**-hard and $B \in \mathbf{NP}$.



2.3.1 Boolean formulae and the CNF form.

Some of the simplest examples of **NP**-complete problems come from propositional logic. A *Boolean formula* over the variables $u_1, \ldots, u_n$ consists of the variables and the logical operators AND ($\wedge$), NOT ($\neg$) and OR ($\vee$); see Appendix A for their definitions. For example, $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$ is a Boolean formula that is True if and only if the majority of the variables a, b, c are True. If φ is a Boolean formula over variables $u_1, \ldots, u_n$, and $z \in \{0,1\}^n$, then $\varphi(z)$ denotes the value of φ when the variables of φ are assigned the values z (where we identify 1 with True and 0 with False). A formula φ is *satisfiable* if there there exists some assignment z such that $\varphi(z)$ is True. Otherwise, we say that φ is *unsatisfiable*.

A Boolean formula over variables $u_1, \ldots, u_n$ is in *CNF form* (shorthand for *Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations. For example, the following is a 3CNF formula:

$$(u_1 \vee \bar{u}_2 \vee u_3) \wedge (u_2 \vee \bar{u}_3 \vee u_4) \wedge (\bar{u}_1 \vee u_3 \vee \bar{u}_4)\,.$$

where $\bar{u}$ denotes the negation of the variable u.

More generally, a CNF formula has the form

$$\bigwedge_i \left(\bigvee_j v_{i_j} \right),$$

where each v_{i_j} is either a variable u_k or to its negation $\neg u_k$. The terms v_{i_j} are called the *literals* of the formula and the terms $(\vee_j v_{i_j})$ are called its *clauses*. A kCNF is a CNF formula in which all clauses contain at most k literals.

Theorem 2.10 (Cook-Levin Theorem [Coo71, Lev73])
Let SAT *be the language of all satisfiable CNF formulae and* 3SAT *be the language of all satisfiable 3CNF formulae. Then,*

1. SAT *is* **NP**-*complete.*
2. 3SAT *is* **NP**-*complete.*

2.5 Decision versus search

We have chosen to define the notion of **NP** using Yes/No problems ("Is the given formula satisfiable?") as opposed to *search* problems ("Find a satisfying assignment to this formula if one exists"). Clearly, the search problem is harder than the corresponding decision problem, and so if $\mathbf{P} \neq \mathbf{NP}$ then neither one can be solved for an **NP**-complete problem. However, it turns out that for **NP**-complete problems they are equivalent in the sense that if the decision problem can be solved (and hence $\mathbf{P} = \mathbf{NP}$) then the search version of any **NP** problem can also be solved in polynomial time.

Theorem 2.19
Suppose that $\mathbf{P} = \mathbf{NP}$. *Then, for every* **NP** *language* L *there exists a polynomial-time TM* B *that on input* $x \in L$ *outputs a certificate for* x.

That is, if, as per Definition 2.1, $x \in L$ *iff* $\exists u \in \{0,1\}^{p(|x|)}$ *s.t.* $M(x,u) = 1$ *where* p *is some polynomial and* M *is a polynomial-time TM, then on input* $x \in L$, $B(x)$ *will be a string* $u \in \{0,1\}^{p(|x|)}$ *satisfying* $M(x, B(x)) = 1$.

To solve the search problem for an arbitrary **NP**-language L, we use the fact that the reduction of Theorem 2.10 from L to SAT is actually a *Levin* reduction. This means that we have a polynomial-time computable function f such that not only $x \in L \Leftrightarrow f(x) \in \mathsf{SAT}$ but actually we can map a satisfying assignment of $f(x)$ into a certificate for x. Therefore, we can use the algorithm above to come up with an assignment for $f(x)$ and then map it back into a certificate for x. ■