# NOTES IN COMPLEXITY THEORY

Bruce Shepherd

Graham Brightwell

Centre for Discrete and Applicable Mathematics
London School of Economics
November 26, 1998

# 1 Introduction

Which of the following is the most difficult task?

- Running fifty times around Hyde Park.
- Transporting an IBM PC 286 by foot to Imperial College.
- Climbing the Zugspitze.

Since some of us are in poor cardio-vascular condition we may immediately gravitate to the second or third options. Others of us may be afraid of gravity itself and will refuse option three. The question is virtually impossible to answer because of subjective preferences but what if we restricted ourselves to asking the question

**How hard is it to do something?**

for a machine such as a computer.

Complexity Theory is about (i) formulating this as a meaningful question and (ii) answering the question in certain sensible instances.

NATURAL QUESTIONS:

**A.** What is "something"?
**B.** What is "hard"?

Some examples of "somethings" may include

- Add 1 to an integer $x$.
- Find the derivative of a polynomial $p(\cdot)$.
- Find a shortest tour which visits each city in a collection $\mathcal{S}$ of world capitals, exactly once.

Different alternatives for measuring "how hard" could include:

- How long does it take to perform the task?

- How much space (disk space for example) does it take?
- How long must we keep the phone line open?

which leads to an even more general *bottom line*

- How much does it cost?

To be able to answer such questions rigorously, complexity theorists formulate the problem more mathematically. Let us consider a new question.

**For a given, fixed, function $F$: How "hard" is it to evaluate $F$?**

Relating this to our previous examples, note that we could choose as our function $F$, any one of the following:

- $F : \mathbf{N} \to \mathbf{N}$ such that $F(x) = x + 1$.
- $F(p) = p'$ for a polynomial $p$.
- $F(M) = l$ where $M$ is an $\mathcal{S} \times \mathcal{S}$ nonnegative, symmetric matrix of distances between pairs of cities in $\mathcal{S}$, and $l$ is the length of a shortest tour through these cities.

This leads to even more questions...

1. What can the range and domain of $F$ be?
2. Doesn't the answer to "how hard" depend on our resources?
   E.g. Do we have a Cray or an abaccus?
3. Does the answer not also depend on *how* we evaluate $F(x)$ since there may be more than one method?
4. The answer also seems intuitively to depend on the input to $F$?
   For example, if $F(x) = x+1$, then one would suspect that $F(99999)$ is harder to evaluate than $F(1)$. Thus our answers to "how hard" should take into account the size of the input.

We will take several lectures to adequately address all of these questions but let us make some initial inroads.

**Some Answers**

**1.** Computers typically take input as strings of *binary bits* (i.e., 0's and 1's) and thus the most sensible and simplest assumption to make is that the range of our functions $F$ will be of this form (strings of 0's and 1's). This is not very limiting since essentially any form of input (integers, rational numbers, city names etc.) can be encoded in this form. And it has the advantage of giving a trivial means by which we measure the *size* of an input: namely that **the size of a string $x$ will be the number of bits in the string**.

**2. Usual Model:** assume an "idealised" computer.

**3. The approach:** Think of programs (i.e., implementations of algorithms) running on the "idealised computer". We then ask how many "steps" this computer takes in running the "shortest" such program which computes $F$. This is rationalised by the fact that most forms of "cost" of performing a function usually depend on the length of time for which a computing resource is required. This in turn is directly proportional to the number of basic operations or *steps* that the computer must perform.

**4.** Suppose that we have a program which computes the function $F(x) = x + 1$ on our idealised computer. Taking into account the size of input into our measure of the running time (i.e., number of steps taken) of our algorithm is just to say that we seek answers of the form:
- if the input $x$ is of length $n$, then $x + 1$ is computed in at most $5n$ steps.

and

- there is no program which can guarantee for every $x$, to compute $x + 1$ in at most $2n + 1$ steps.

Our objective in measuring speed should be to give answers which are not sensitive to the continual improvements in computer processing power. That is, we really want to measure the intrinsic difficulty of problems.

We do not want to be told that if we make an upgrade from our PC386 to a PC486, then suddenly the problem of adding 1 to an integer will become easy instead of hard.

This means that for our purposes, constant factors in the running time of a program or algorithm should be ignored. In other words, $5n$ and $2n+1$ are the same for us since they are both *linear* in the length of the input.

# Preliminary notes, taken from last year's course, for Oct 15 and Oct 22/23, 1998

## ALGORITHMS AND LANGUAGES

| | |
|---|---|
| Garey and Johnson | *Algorithms are general, step-by-step procedures for solving problems* |
| Wilf | *An algorithm is a method for solving a class of problems on a computer* |
| Biggs | *In order to give a completely satisfactory definition of what we mean by an algorithm we should delve quite deeply into the realms of mathematical logic. ...Roughly speaking, an algorithm is a sequence of instructions* |
| Cutland | *An algorithm is a mechanical rule or automatic method, or programme for performing some mathematical operation* |
| Hamilton | *An algorithm is an explicit effective set of instructions for a computing procedure (not necessarily numerical) which may be used to find the answers to any of a given class of questions* |
| Brightwell | *A procedure so clearly defined that even a computer can carry it out* |

**Conclusion:** Difficult to say without referring to some idealised computer.

For the moment, assume we have such a computer in mind. Then *an algorithm for us will be a program for that computer.* With a view to defining formally our idealised computer, we make even more precise the essential problem we want to solve.

Let $\Sigma$ denote any finite alphabet, i.e., finite set of symbols (typically $\{0,1\}$). Let $\Sigma^*$ denote the set of finite strings (including the empty string) of characters from the alphabet $\Sigma$.

**Definition 1.1** *A* <u>*language*</u> *over* $\Sigma$ *is a subset of* $\Sigma^*$.

**Definition 1.2** *The decision problem for a language $\mathcal{L}$ is that of determining whether a given input string is in $\mathcal{L}$.*

**Definition 1.3** *An algorithm solves the decision problem for $\mathcal{L}$ if (i) whenever the input $x$ is in $\mathcal{L}$, the algorithm outputs YES and (ii) whenever the input $x$ is not in $\mathcal{L}$, the algorithm outputs NO.*

As a brief aside, we mention that an algorithm is said to *recognise* $\mathcal{L}$, if it terminates with output YES if and only if the input $x$ is in the language. Clearly, if there exists an algorithm which solves the decision problem for a language, then there is also an algorithm which recognises it. The converse, however, is false. One natural question is:

**Question 1** *Can all languages be recognised?*

We will see that the answer to this question is no. There are thus two possible types of questions regarding the hardness of computing languages:

1. Which languages can be recognised at all?
2. Which languages can be recognised "fast"?

This gives different interpretations of what it means for a problem to be *intractable*. For the most part, this course focuses on questions of the second variety, i.e., which decision problems can be solved by algorithms with a fast running time? We delay the precise definition of 'fast' until later and turn first to the precise definition of our idealised computer.

## 2 Turing Machines: A Model of Computation

We now introduce a computer which goes by the name *Turing Machine*, named after the $20^{th}$ century mathematician Alan Turing. Its description is sufficiently simple to allow us to mathematically analyze its behaviour. At the same time, it is sufficiently powerful to be able to perform any task that any other computer can (although it may take a "little" longer).

A *Turing machine* (TM) consists of:

(a) a **tape**,

(b) a **tape head**,

(c) a **finite state component**.

The *tape* is a 2-way infinite strip divided into squares, one associated with each integer $\ldots -2, -1, 0, 1, 2, 3 \ldots$. Each square may have any one of a finite set $\Gamma$ of *tape symbols* written in it. Additionally it is assumed that there is a distinguished *blank symbol* $\# \in \Gamma$ and that at any time the tape contains at most a finite number of squares with non-blank symbols (i.e., the tape is padded with blanks in both infinite directions).

The *tape head* is alway positioned over some square, called the *current square*, of the tape. It is capable of

- reading the symbol in that square,
- writing a symbol in that square (i.e., overwriting the previous symbol),
- moving one square to the left or right.

The *finite state component* is used for storing: (i) A distinguished element of a finite list of *states*. This element is the *current state* of the machine. (ii) A *finite state control* which consists of a finite set of "rules" for progressing from one current state to the next. When the Turing machine is "turned on" the machine obeys these rules until it reaches one which halts the machine. Note that although the contents of the finite

state component is finite, it is neither fixed nor bounded and in particular we have the ability to change it.
This now specifies the "hardware". Note that there exist many Turing machines - one associated with each set of tape symbols $\Gamma$. For our purposes we prefer to think of a single fixed machine and so unless stated otherwise we assume that $\Gamma = \{0, 1, \#\}$.

Speaking informally, a *program* for a Turing machine tells the machine what to do. It includes a list of possible machine states which contains a specified start state $q_0$ and two distinguished *halt states* $q_N, q_Y$ which indicate that program execution has terminated. At each step, the program specifies what the machine is to do depending on two things: (i) which state the machine is currently in and (ii) which symbol is currently being scanned by the tape head. The program must "tell" the machine

- what to write in the current square
- where to move the tape head (left, right, stationary) for the next step, and
- which state to go to.

A Turing machine has the ability to do these three actions simultaneously and this is considered its *basic operation.* Each time the computer performs this operation, the algorithm is said to have taken a *step.*

More formally, a *program* is an ordered triple $(\Sigma, Q, \delta)$ which specifies the following:

1. A subset $\Sigma \subseteq \Gamma - \{\#\}$ of *input symbols.*
2. A finite set of *states* $Q$ which contains a distinguished *start state* $q_0$ as well as *halt states* $q_N$ and $q_Y$.
3. A *transition function* $\delta : (Q - \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{L, R, -\}$.

The running of a program is as one might expect. An input string $x \in \Sigma^*$ is written in the squares $1, 2, \ldots, n$ and initially all other squares contain the blank symbol #. The tape head starts at position 1 and the machine

is initially in state $q_0$ (i.e., the finite state component contains the current state $q_0$). The machine then obeys the transition function until it enters state $q_Y$ or $q_N$ (Yes or No) at which time computation halts.

Recall that we identify algorithms as being programs for our idealised computer and so henceforth we make no distinction between the terms algorithm and Turing machine program.

Let's consider now an example. Unless otherwise stated, our programs will have input alphabet $\{0, 1\}$.

**Algorithm 2.1** *Our first Turing machine program* $M = (\{0, 1\}, Q, \delta)$ *has finite state set* $Q = \{q_0, q_1, q_Y, q_N\}$ *and transition function* $\delta$ *defined by the following table:*

| *State* | *Symbol* | $\rightarrow$ | *State* | *Symbol* | *Direction* |
|---|---|---|---|---|---|
| $q_0$ | *0* | | $q_0$ | *0* | *R* |
| $q_0$ | *1* | | $q_1$ | *1* | *R* |
| $q_0$ | # | | $q_N$ | | |
| $q_1$ | *0* | | $q_1$ | *0* | *R* |
| $q_1$ | *1* | | $q_N$ | | |
| $q_1$ | # | | $q_Y$ | | |

Note that in Algorithm 2.1, if the input is 000100, then the tape head moves three times to the right remaining in state $q_0$ and the head is then scanning the input symbol '1'. At this point the program "instructs" the machine to enter state $q_1$. The tape head then continues moving to the right for the next three steps of the program execution until the current square contains #. The program then halts execution and the machine is left in state $q_Y$. If the input string were to contain a second 1-bit, then note that the machine would enter state $q_N$ and execution would immediately halt.

**Remark:**

For those with a familiarity with computer programming, the finite state component of a Turing machine may be roughly associated with the code of some program. More precisely, the list of states will correspond to such a code and each individual state then corresponds to a single instruction. Consider the following pseudo-code:

| | |
|---|---|
| $q_0$ | $i \leftarrow 1$ |
| $q_1$ | **while** $i \leq 6$ |
| $q_2$ | $i \leftarrow i + 1$ |
| $q_3$ | **end while** |
| $q_4$ | **write "Goodbye cruel world"** |

If the five righthand instructions were run on some computer then the *while loop* would be performed 6 times until $i = 7$ at which point the execution branches to the *write statement*. One could think of this as a program which 'cycles' through the states $q_1, q_2$ and $q_3$ until a certain 'triggering event' occurs, namely that $i$ gets the value 7, at which point the machine moves from state $q_1$ to state $q_4$ instead of moving to $q_2$ for a seventh time. The main point is that **the change of states is dependent on the contents of memory** (in this case, the memory contents of our variable $i$) in the same way that the change of states in a Turing machine depends on the contents of the current square which is part of its memory (the tape).

For a Turing machine it is the transition function $\delta$ which defines what sort of instruction a state $q_i$ is, i.e., what it will 'do' depending on memory. Thus choosing the statements and their order in some programming code is in essence defining a transition function for the machine on which we will run the code. In the above example, if we replaced the second statement by *while* $i > 2$, then the transition function would be substantially different since execution would no longer branch to state $q_2$ whenever the memory location for variable $i$ holds the value 1.

An algorithm $M$ is said to *accept* an input string $x \in \Sigma^*$ if and only if the TM halts in state $q_Y$ when the program is run with input $x$. The language *recognised* by the algorithm is $L_M = \{x \in \Sigma^* : M \text{ accepts } x\}$. It is not hard to see that the language recognized by Algorithm 1 is $L = \{x \in \{0,1\}^* : x \text{ contains exactly one } 1\}$. We re-emphasize that recognition of $L \subseteq \Sigma^*$ does not require $M$ to halt on every possible input string. For this example, however, $M$ does indeed halt for every string in $\Sigma^*$ and so additionally *$M$ solves the decision problem for $L$.*

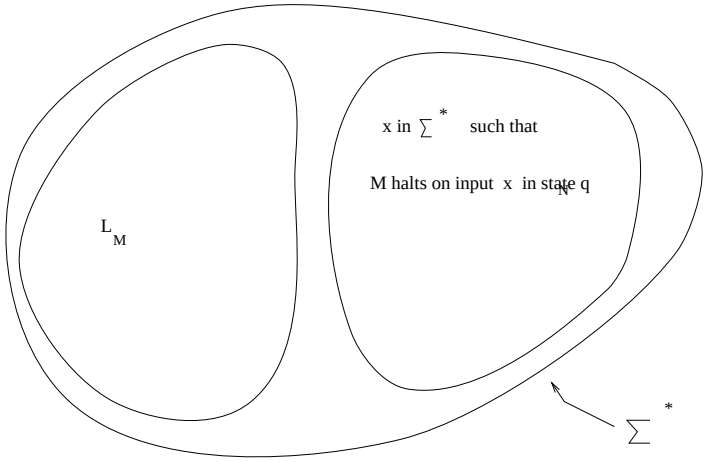


Figure 1:

## Further Examples and Representations of Turing Machine Programs

Since the next state of the machine depends only on the current state and the symbol in the current square, it is usually more useful (and more compact) to represent the transition function of an algorithm as a $Q \times \Gamma$ matrix called a *program table* (as in the following example).

**Algorithm 2.2** $Q = \{q_0, q_1, q_2, q_Y, q_N\}$

| $q$ | *0* | *1* | # |
|---|---|---|---|
| $q_0$ | $(q_0, 0, R)$ | $(q_0, 1, R)$ | $(q_1, \#, L)$ |
| $q_1$ | $(q_2, \#, L)$ | $(q_N, \#, L)$ | $(q_N, \#, L)$ |
| $q_2$ | $(q_Y, \#, L)$ | $(q_N, \#, L)$ | $(q_Y, \#, L)$ |

One routinely checks that the language accepted by the above algorithm is $L = \{x \in \{0,1\}^* :$ nonempty strings s.t. neither of the two rightmost symbols of $x$ is a $1\}$. This seems a language of marginal interest but suppose you needed to solve the following problem: given an integer $n$, determine whether $n$ is divisible by 4 or not. Algorithm 2.2 essentially solves this problem in the sense that the strings in $L$ are precisely those which are binary representations of integers divisible by four.

A *palindrome* is a string which when read in reverse order remains the same string. Consider the problem of designing an algorithm which solves the decision problem for the language consisting of all palindromes. Design your algorithm and then check it with the following:

**Algorithm 2.3** $Q = \{q_0, q_1, \ldots, q_5, q_Y, q_N\}$.

| $q$ | *0* | *1* | # |
|---|---|---|---|
| $q_0$ | $(q_1, \#, R)$ | $(q_2, \#, R)$ | $q_Y$ |
| $q_1$ | $(q_1, 0, R)$ | $(q_1, 1, R)$ | $(q_3, \#, L)$ |
| $q_2$ | $(q_2, 0, R)$ | $(q_2, 1, R)$ | $(q_4, \#, L)$ |
| $q_3$ | $(q_5, \#, L)$ | $q_N$ | $q_Y$ |
| $q_4$ | $q_N$ | $(q_5, \#, L)$ | $q_Y$ |
| $q_5$ | $(q_5, 0, L)$ | $(q_5, 1, L)$ | $(q_0, \#, R)$ |

Yet another way of representing a Turing machine program (and usually the easiest) is by means of a *flow diagram*. Such a diagram contains a "node" $v_q$ for each state $q$ of the program and for each possible symbol on the current square there is a "directed arc" $a_{q,v}$ with its tail at $v_q$ and its head pointing at the node which corresponds to the next state given by the transition function. In addition, the arcs are labelled with the instructions for the tape head so that an arc labelled $0 : 1, R$ whose tail is at the node for state $q_0$ and head is at $q_2$ informs us that if we ever scan the symbol 0 while in state $q_0$, then 1 is written on the current square, the tape head is moved to the right and the current state becomes $q_2$. A flow diagram for Algorithm 2.1 is depicted in Figure 2.

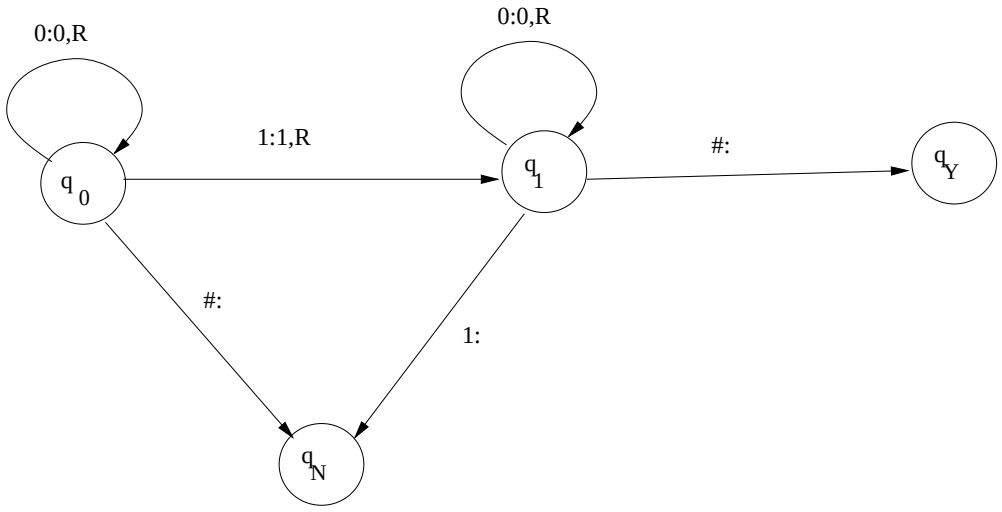


Figure 2:

## Exercises

[**2.1**] Write Algorithm 2.3 using a flow diagram.

[**2.2**] Write programs (table or flow diagram) to recognise the following languages:

- $\{x \in \{0, 1, \}^* :$ $x$ has length at least 5$\}$
- $\{x \in \{0, 1, \}^* :$ $x$ does not have a pair of consecutive 1's $\}$.

# Preliminary notes for Oct 22 and Oct 29, 1998

## Integer Maps

Suppose that $M$ is any algorithm. We say that *$M$ computes the (partial) function* $g_M : \Sigma^* \to \Gamma^*$ where $g_M(x)$ is defined by running $M$ with input $x$: if $M$ does not halt, then $g_M(x)$ is undefined; if $M$ does halt, then $g_M(x)$ is obtained by forming the string of symbols in the squares $1, 2, 3, \ldots$ up until the first blank symbol (after $M$ has halted). We say that $g_M(x)$ is the *output* of *$M$ on input $x$*.

This gives rise naturally to the definition of computing functions with domain and range $\mathbf{N}$. Let us denote by $b_n$ the binary representation of $n$ and similarly denote by $m_n$, its monadic representation, i.e., the string of $n$ ones. Then $M$ is said to *compute $f : \mathbf{N} \to \mathbf{N}$ under binary* (respectively *monadic*) *notation* when for each natural number $n$, if $M$ is given the input string $x = b_n$ (respectively $x = m_n$), then $g_M(x)$ is the binary (respectively monadic) representation of $f(n)$ if $f(n)$ is defined, and if $f(n)$ is not defined, then $g_M(x)$ is also not defined.

**Definition 2.1** *For a given algorithm $M$, its (binary) integer map, denoted $f_M$, is the function satisfying:*

$$f_M(n) = \begin{cases} \textit{undefined} & \textit{if } M \textit{ does not halt when given } b_n \textit{ as input} \\ k & \textit{if } g_M(b_n) \textit{ is the binary representation of } k. \end{cases}$$

When it is clear from the context, we sometimes say that an algorithm takes computes $f(n)$ on input $n$ in order to refer more directly to its integer map. It should be understood of course that this is just a shorthand convenience for saying that the algorithm is run with the input string $b_n$ on its input tape starting at square one. We close by mentioning that these concepts may be naturally extended to integer valued functions of two variables $f(m, n)$ (or $k$ variables $f(n_1, n_2, \ldots, n_k)$) as long as we specify some convention for reading inputs $m, n$. For example, under monadic notation the squares $1, 2, \ldots$ could contain $m$ 1's followed by a 0 and then $n$ 1's followed by a #.

## 3 What Qualifies as Computation? Church's Thesis

Speaking broadly, Church's Thesis asserts that any (imaginable) form of computation could be simulated by a Turing machine (although it may take a lot longer). In believing this (which we will) we are assuming that these quite rudimentary machines capture the essence of what we intuitively understand as a mechanical routine.

A function $f : \mathbf{N} \to \mathbf{N}$ is *Turing-computable* if it is the integer map of some Turing machine program.

**Exercise 3.1** *Explain why the set of Turing-computable functions does not change if we alter Definition 2.1 of integer maps to be those computed under monadic notation.*

It is more customary to state Church's Thesis as:

**Church's Thesis** *Any "computable" function is Turing-computable.*

Of course you might say that it is doubtful that this would admit a mathematical proof. And so it would seem since how could we possibly account for any possible mode of computing including as-yet undiscovered schemes of computation. On the other hand, if false, then it would be refutable since you could show me one of your 'schemes' and persuade me that it represents "computation". Then you would convince me that there was a Turing-noncomputable function $f$ which was computable by your methods.

If we cannot give a proof of Church's Thesis then we are left with the alternative of giving supporting evidence. This is usually done by considering ostensibly more powerful models of computation and then showing that they could be simulated by a Turing machine. One such example is a Turing Machine in which the tape has $k$ tracks: *$k$-track* Turing machine - see Figure 3.

It is straightforward to see that such a machine with tape alphabet $\Gamma$ could be simulated by a Turing machine with tape alphabet $\Gamma^k$ ($k$ time cartesian product of $\Gamma$) since the $k$-track Turing machine makes its

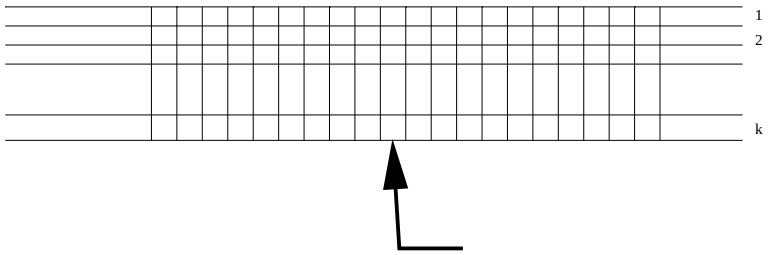


Figure 3:

next step depending on the contents of the $k$-tracks which can simply be thought of as a $k$-tuple of symbols from $\Gamma$.

Another typical model is a *$k$-tape Turing machine*. This consists of $k$ tapes and $k$ tape heads together with a finite state component which now also indicates which of the $k$ tapes is currently *active*. A program for a $k$-tape machine now specifies a transition function

$$\delta : (Q - \{q_N, q_Y\}) \times \underbrace{\Gamma \times \Gamma \ldots \times \Gamma}_{k \text{ times}} \times \{1, 2, \ldots, k\} \to Q \times \Gamma \times \{L, R, -\} \times \{1, 2, \ldots, k\}.$$

This determines the next current state, the next active tape, and what to write and which way to move on the currently active tape. This machine can quite obviously compute any function that a normal Turing machine can; it is not immediately obvious that the converse is also true.

**Exercise 3.2** *Show that any function which can be computed by a $k$-tape Turing machine is also Turing-computable.*

**Hint:** Show that a $k$-tape Turing machine can be simulated by a $2k$-track machine, two tracks simulating each of the $k$ tapes. Each tape will have a track which holds the corresponding contents of the tape in the $k$-tape machine, and the other holding all blanks except for a single marker which indicates the tape head position for the corresponding tape.

A more pertinent example would be to consider a so-called real computer. By this we mean a machine which has an unlimited amount of memory consisting of *registers* $R_1, R_2, \ldots$ which may contain any integer. The computer also has the capability of performing basic arithmetic operations $+, -, \times, /$ on two of these registers and storing the result into a third: $R_2 \leftarrow R_1 + R_k$. Computers in daily life do not have unbounded amounts of memory and their registers cannot store arbitrarily large integers but these bounds are simply a function of the present state of technology and we are concerned with a theory which is independent of this and so we adopt the above formulation.

**Exercise 3.3** *Describe how to simulate a real computer by a 4-tape Turing machine.*

## 4 Intractability I: Noncomputability

Having given the definition of algorithm, it is simple to answer the following question.

**How many "different" $k$-state algorithms are there?**

Of course, we are most interested in what an algorithm actually 'does' or 'computes' for us and this is independent of how we name its states. Thus to sensibly answer this question we assume that the names of states for an algorithm are drawn from the set of natural numbers and that in fact an algorithm with $k$ states will have states named $1, 2, \ldots, k$. We also may assume that $q_0$ is understood to be State 1 and that $q_N, q_Y$ are represented by states $k-1$ and $k$ respectively. This amounts to saying that the 'effects' of running a program are determined only by its transition function. The number of such functions with $k$ states (i.e., with domain $\{1, \ldots, k-2\}$) is easily computed. This is the same as counting the number of distinct $\{1, \ldots, k-2\} \times \Gamma$ program tables (remember there is no action to be taken in either of the two halting states). Such a table is determined by defining for each of its $(k-2)|\Gamma|$ entries, an action from the set $\{1, \ldots, k\} \times \Gamma \times \{L, R, -\}$.

| | 0 | 1 | # |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| ⋮ | | ⋮ | |
| k-2 | | | |

The number of such choices for each entry is evidently $3k|\Gamma|$. Since this choice is made for each entry of the table, the number of possible program tables is $(3k|\Gamma|)^{(k-2)|\Gamma|}$ and so this is the number of $k$-state algorithms (under our convention for naming states). If our Turing machine uses only the symbols $0, 1, \#$, then the number of algorithms with $k$ states is $(9k)^{3(k-2)}$.

Consider now listing (in some logical order) the 3-state algorithms, and then adding a list of the 4-state algorithms, then the 5-state algorithms, and so on. In the previous paragraph, we showed that each of these sublists is finite and so this gives rise to a proper list $\mathcal{A} = M_1, M_2, \ldots, M_i, \ldots$ of *all* algorithms. This shows that:

**The set of algorithms for a Turing machine is denumerable.**

We can now readily see that the answer to Question 1 in Section 1 is no. The reason is that the collection of algorithms is denumerable whereas the collection $\mathcal{L}$ of languages over $\{0, 1\}$ is uncountably infinite. Thus there are 'too few' algorithms to be able to compute all languages.

**Exercise 4.1** *Show that $\mathcal{L}$ is uncountably infinite.*

Exactly the same reasoning shows that there is some function $f : \mathbf{N} \to \mathbf{N}$ which is not computable by any Turing machine program.

Lets return briefly to our list of algorithms. We have in fact calculated the exact length of each of the sublists of $k$-state algorithms. So if we wanted to find the $k$-state Turing machines in $\mathcal{A}$, we would move to the $p_k^{th}$ element of the list, where $p_3 = 1$ and for $k \geq 4$, $p_k = 1 + \sum_{i=3}^{k-1} (9i)^{3i-6}$. Moreover, it is possible to make this ordering so that for any integer $n$ we could determine precisely what the $n^{th}$ algorithm is! That is we could construct an algorithm which, given an integer $n$, writes out the transition function for the $n^{th}$ algorithm in the list. This can be done, for example, by making the list in a lexicographic order (see Solution 4.2). We will assume that we have fixed such an ordering $\mathcal{A}$ and for an algorithm $M$, its *Turing number,* denoted $\sigma(M)$, is the position where it appears in $\mathcal{A}$. Note that there is some redundancy in the list in the sense that two algorithms may actually compute the same function. (Why?)

The argument that there exist functions which are not Turing computable is slightly unsettling because it only guarantees that there *exists* some uncomputable function (because there are more functions than there are algorithms to compute them). Is it possible to actually find one of these *uncomputable functions*?

The answer is yes. Let $f_n$ be the integer map of the algorithm with Turing number $n$ and let $u : \mathbf{N} \to \mathbf{N}$ be the function:

$$u(n) = \begin{cases} 1 & \text{if } M_n \text{ does not halt when given input string } b_n \\ f_n(n) + 1 & \text{otherwise} \end{cases}$$

**Claim:** *$u$ is not computable.*

**Proof:** If $u$ were computable, then by Church's Thesis, there is some algorithm $M$ which computes it. That is to say, since $u$ is defined for every natural number, $u(n) = f_M(n)$ for all $n \in \mathbf{N}$. Also from the previous paragraphs, $M$ has a Turing number, say $p$, and so $f_M \equiv f_p$ and hence

$$u(n) = f_p(n) \quad \forall n \in \mathbf{N}. \tag{1}$$

In particular $u(p) = f_p(p)$ and so $f_p(p)$ is defined. Thus $M_p$ halts when given input $p$ (or more precisely when given its binary representation $b_p$). But the definition of $u$ then states that $u(p) = f_p(p) + 1$, contradicting (1). Thus our assumption that $u$ was computable must have been false. □

### Undecidability and the Halting Problem

To simplify things in this section, we usually speaking about the actions of an algorithm 'on input $n$', where more precisely it is understood that we mean 'when given the input string $b_n$'.

A function $f : \mathbf{N} \to \{0, 1\}$ or equivalently a decision problem, is *undecidable* if $f$ is uncomputable. Define the *self-halting function* $s : \mathbf{N} \to \mathbf{N}$ as

$$s(n) = \begin{cases} 1 & \text{if } M_n \text{ does not halt on input } n \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $s$ is computable and $H$ is an algorithm which computes $s$. Then consider a new algorithm $M$ whose definition should be clear from Figure 4.

Then for any integer $n$, $M$ first simulates $H$ and then either goes to state $q_Y$ or state $q^\infty$. $H$ computes $s$ and since $s$ is defined for each $n$,

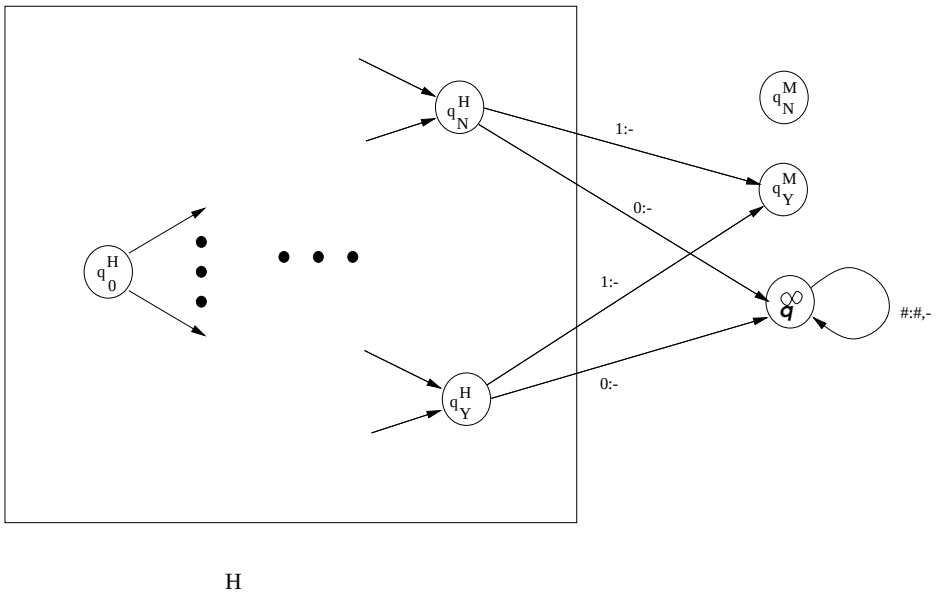


Figure 4: *Flow Diagram for M*

this means that computation will always halt in state $q_Y^H$ or $q_N^H$. At this point, $M$ will either (a) enter state $q_Y$, if $H$ wrote 1 on the tape, or (b) write # and enter state $q^\infty$. In the latter case, note that $M$ will then repeatedly loop in state $q^\infty$ forever. Thus $M$ will halt on input $n$ if and only if $H$ outputs 1 when given input $n$. By definition. this happens if and only if $M_n$ does not halt when given input $n$. Thus

$$M \text{ halts on input } n \Leftrightarrow M_n \text{ does not halt on input } n \qquad (2)$$

But $M$ is an algorithm, so let $k$ be its Turing number and consider plugging $k$ in for $n$ in equation (2). Then $M$ halts on input $k$ if and only if $M_k$ does not halt on input $k$. But $M = M_k$ and so we have the absurd: $M$ halts if and only if it doesn't! This contradiction no refutes that $s$ is a computable $0-1$ function, i.e., $s$ is undecidable.

This is an interesting point as it implies that there is no algorithmic way to determine whether computation will halt for a given program and input. We can do still better than this. We can show that there is a **specific** program $U$ for which it is even undecidable to determine whether $U$ itself halts on a given input.

### Universal Turing Programme

We define the function

$$f(m,n) = \begin{cases} \text{undefined} & \text{if } M_m \text{ does not halt when given input string } b_n \\ f_m(n) & \text{otherwise.} \end{cases}$$

Is $f(m,n)$ computable?

The answer is yes: by the *universal Turing program* $U$. This is a program which for any input specifying two integers $m, n$, first 'constructs' the Turing program $M_m$ and then simulates $M_m$'s execution with the input $n$. Thus it halts with the value $f_m(n)$ on its tape if and only if $M_m$ would halt on input $n$. How can such a universal algorithm exist? It seems miraculous at first. We ask you to think about how $U$ could be constructed (before referring to the solutions and hints section).

**Exercise 4.2** *Describe how $U$ can be constructed.*

For now, we assume that $U$ does indeed exist and show, using our previous work, that it is undecidable to determine whether $U$ halts on a given input $m, n$. For suppose this is not the case. Then there would be some algorithm $M$ which, given input $n$, halts and outputs 1 if $U$ never halts when given the input pair $(n,n)$ and otherwise $M$ would not halt (or outputs 0 if you prefer). We can use this to to compute $s$ as follows.

1. Simultaneously run $U$ and $M$ with inputs $(n,n)$ and $n$ respectively.
2. If $U$ halts, then this means that $M_n$ would have halted on input $n$ and so we output 0.
3. Otherwise, $U$ would never halt but then $M$ would and its output would be 1.

Thus together, these machines can be used to compute $s$. Now by Church's Thesis, anything computed by two machines could be computed just as well by a single machine and so $s$ would be computable, which we have already seen to be false.

Some other undecidable problems arising in mathematics include:

- Given a polynomial $p : \mathbf{R}^k \to \mathbf{R}$, do there exist integers $x_i$ such that $p(x_1, x_2, \ldots, x_k) = 0$?
- Given a polygon $P$, can $P$ be tesselated to perfectly cover the plane?
- Given a finitely generated group $G$ and a word $\omega = \sigma_1\sigma_2 \ldots \sigma_k$, is $\omega$ the identity of the group?

**Solution 4.2**

The essence of the universal Turing machine is choosing our list $\mathcal{A}$ so that we can algorithmically determine the algorithm with any given Turing number. One such ordering would be a lexicographic ordering. We will place an ordering on the 'actions' in a $k$-state program table as follows. First we assume that the tape symbols and set of tape moves are ordered, e.g., for $\{0, 1, \#\}$ we assume the order $0 < 1 < \#$ and the tape moves are ordered $L < R < -$. Now any action $(a, b, c)$ (where for example $a = 4, b = \#, c = R$) can be ordered relative to any other action $(a', b', c')$ as follows:

$$(a, b, c) < (a', b', c')$$

if either (i) $a < a'$ or (ii) $a = a'$ and $b < b'$ or (iii) $a = a', b = b'$ and $c < c'$. This now gives rise to an ordering of the actions in a program table.

This extends easily to an ordering of the set of algorithms (or program tables) as follows. For an algorithm $A$ and integers $i = 1, \ldots, k-2$; $j = 1, 2, 3$ let $A_{ij}$ denote the action in the $ij$ entry of $A$'s program table ($j = 1, 2, 3$ will correspond respectively to the tape symbols $0, 1, \#$ by convention). Then for two $k$-state algorithms $A, B$ their Turing numbers satisfy $\sigma(A) < \sigma(B)$ if either [1] $A_{11} < B_{11}$ **OR** [2] $A_{11} = B_{11}$ and $A_{12} < B_{12}$ **OR** [3] $A_{11} = B_{11}, A_{12} = B_{12}$ and $A_{13} < B_{13}$ **OR** $\ldots$ **OR**$[3(i-1) + j]A_{lp} = B_{lp}$ for each $l, p$ where $l < i$ or $l = i; p = 1, \ldots, j-1$, and $A_{ij} < B_{ij}$ **OR** $[3(i-1) + j + 1] \ldots$ .

This has now specified an ordering of the $k$-state Turing programs. As we have seen before, we already know the first position of the $k$-state algorithms to be in the position $p_k$ of $\mathcal{A}$. So under our lexicographic

ordering, we could find the algorithm with Turing number $m$ as follows: first look for $p_k$ such that $p_k \leq m < p_{k+1}$. This implies that the algorithm with Turing number $m$ has $k$ states. Next we apply the rules from the preceding paragraph to determine the program table in the $m - p_k$ position amongst the $k$-state tables. This shows how given an integer $m$, we can construct the transition function for the algorithm with Turing number $m$.

This is the meat of what the universal Turing program must do when given input $(m, n)$. It then 'writes down' this program table on its tape and obeys its rules applied to an input of $n$. We have ignored the gory details of how this can all be encoded in a 'mega-transition function', preferring instead to remain in a lofty meta-proof mode.

## 5 Tractability II: Fast or Slow?

### Running Time and Some Basics

The *time complexity function* $T_M(n)$ of an algorithm $M$

$$T_M(n) = \max\{m : \exists x \in \Sigma^* \text{ of length } n \text{ such that } M \text{ runs for } m \text{ steps on input } x\}.$$

It is easily checked that the algorithm of Algorithm 2.1 has time complexity function $T_M(n) = n$. In Section 1, (Answer 4.) we proposed that any measure of speed should be insensitive to constant factors. To this end, we make the following definitions.

**Definition 5.1** *For two nonnegative functions $f, g$ defined on the natural numbers, $f(n)$ is order $g(n)$, written $f(n) = O(g(n))$, if there exists a constant $C$ such that $f(n) \leq Cg(n)$ for all $n \in \mathbf{N}$.*

**Definition 5.2** *An algorithm has running time $O(g(n))$ if there exists a constant $C$ such that for $T_M(n) \leq Cg(n)$ for all $n \in \mathbf{N}$.*

We briefly include two other definitions which we encounter later.

**Definition 5.3**

1. *$f(n) = \Omega(g(n))$ if $\exists \epsilon > 0$ such that $f(n) \geq \epsilon g(n),\ \forall n$.*
2. *$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

### The Class $\mathcal{P}$

A *polynomial time algorithm* (or polytime for short) is one whose running time is order $p(n)$ for some polynomial $p$. Polynomial running time of an algorithm was first proposed in the 1960's by Jack Edmonds as the criterion for an algorithm to be "fast" or "efficient" or as he stated it: "good". We denote by $\mathcal{P}$, the following class of decision problems $D(L)$ for a language $L$:

$$\{D(L) : \exists \text{ a polytime algorithm which solves } D(L)\}$$

This now leads to a new view which sees this class $\mathcal{P}$ of (decision) problems to be the 'tractable' ones. The justification for this can be understood from the following data which appears in tables given by Garey and Johnson.

| Time | $n = 10$ | $n = 20$ | $n = 40$ | $n = 60$ |
|---|---|---|---|---|
| $n$ | .00001 sec | .00002 sec | .00004 sec | .00006 sec |
| $n^2$ | .0001 sec | .0004 sec | .0016 sec | .0036 sec |
| $n^3$ | .001 sec | .008 sec | .064 sec | .216 sec |
| $n^5$ | .1 sec | 3.2 sec | 1.7 min | 13 min |
| $2^n$ | .001 sec | 1 sec | 12.7 days | 366 centuries |
| $3^n$ | .0059 sec | 58 min | 3855 centuries | $1.3 \times 10^{10}$ millenia |

For many problems, we have a fixed upper bound on the time in which to finish a computation, e.g., within the ten seconds between bids on the stock exchange, before the end of a quarter for the payroll department, or by the time I retire. The second table shows the impact of improving technology on the maximum size of a problem which is sovable in some bounded length of time. Note the paltry improvements in the last row whereby increasing our computer's speed by a thousand times will only allow an $O(3^n)$ algorithm to solve slightly larger instances of a problem. This indicates how using polynomiality as a measure of fastness is consistent with our earlier prescribed aims (see Answer **3.** on Page 2).

| Time | With Processing speed $P$ | with speed $100P$ | with speed $1000P$ |
|---|---|---|---|
| $n$ | $N_1$ | $100N_1$ | $1000N_1$ |
| $n^2$ | $N_2$ | $10N_2$ | $31.6N_2$ |
| $n^3$ | $N_3$ | $4.64N_3$ | $10N_3$ |
| $n^5$ | $N_4$ | $2.5N_4$ | $3.98N_4$ |
| $2^n$ | $N_5$ | $N_5 + 6.64$ | $N_5 + 9.97$ |
| $3^n$ | $N_6$ | $N_6 + 4.19$ | $N_6 + 6.29$ |

## Some Caveats

There are some practical considerations that one should beware of before buying the whole polynomiality line, hook and sinker. Suppose for example that $M$ has running time $O(p(n))$ for some polynomial $p$.

**1.** *Large Exponents*:
What if $p(n) = n^{1000000^{10000}}$? This algorithm will still outperform any exponential algorithm in the long run. Unfortunately the long run will be for $n$ equal to the number of electrons in the universe. Thus the algorithm would be of little practical use in solving problems.

**2.** *Large Constants*:
A similar problem arises even when $p(n)$ has a small exponent. For example $M$ may be a linear time ($O(n)$)) algorithm but this only implies that its running time is bounded by $Cn$ for some constant $C$. If $C = 100000^{1000000000}$, then potentially we could not even run our algorithm on an instance of size $n = 1$ in our lifetime.

Fortunately in practice, it has been the case that once a polytime algorithm has been found, it is only a matter of time before one is developed (provided there is interest to implement such an algorithm in the computer science community) which has small constant (i.e., $C \leq 50$) and exponent (i.e., at most 3).

**3.** *The maxim 'All exponential algorithms are bad' is bad*
Curiously enough, there are certain exponential algorithms which preform extremely well in practice. Most notably, the simplex algorithm. It has been shown (for each pivot rule) that there exist linear programming problems for which the simplex algorithm performs badly. That is to say, its time complexity function is exponential. It seems, however, that the examples where it runs slowly are quite pathological and essentially never arise.

Another phenomenom has been the existence of algorithms with running time, say, $O(n^{log(log(log(n)))})$. Such an algorithm is not polynomial since the exponent grows with $n$, however it grows so slowly that its running time may be suitable for solving reasonably large instances of real-world problems.

# Notes for Nov 5, 1998

## 6 The Class $\mathcal{NP}$

The next important class we consider is called $\mathcal{NP}$ which means **nondeterministic polynomial** and **does not mean** nonpolynomial. We give two separate definitions of this class of decision problems, but first let us give an intuition behind the class.

This can be described in terms of King Arthur and his magician Merlin. Suppose that Arthur poses a Yes-No question to Merlin. To answer the question seems to require an impossible amount of toil. On the other hand, if the answer is Yes, there is a piece of evidence, or *certificate*, which can be used (even by mortals such as Arthur) to quickly verify the Yes-ness. Merlin is able to find this certificate with his magic and once found, he offers it to the King who then runs through a simple checking routine to see that Merlin is right and the answer is indeed Yes. Note we assert nothing about what happens when the answer is No. In that case there need not exist any such certificate or checking procedure.

The idea embodied in the above story is: *a decision problem is nondeterministic polynomial if there is an easy way to see that the answer is YES <u>when</u> the answer is YES.*

To take this further, let us consider the *Travelling Salesman Problem* (TSP): given a set of cities, find a tour which visits each city exactly once and has minimum total length. For example, suppose you have an assembly line for printed circuit boards and a robot which welds or shoots a laser at some pre-specified locations on each board. Suppose that the number of locations is 1000 (a modest amount in practice) and in order to meet a company schedule, your boss asks *Can we complete the robot phase in under 30 seconds per board?* You take into account the robot speed and laser-shooting time and determine that this is equivalent to requiring a TSP tour of length no more than 5 meters. You also notice

that checking each possible tour amounts to (i) fixing some city from which to start and then (ii) choosing a second city (999 choices) and a third city (998 choices) and so on. Thus the number of tours is 999! and since $n! \geq 2^n$ for all $n \geq 4$, you become quite disturbed when you look at the previous tables to see that $2^{60}$ is already prohibitively large, let alone $2^{999}$.

So you take the circuit board home and start looking for tours randomly and then greedily, but with no luck. Eventually you start to apply some ideas you heard in a combinatorial optimisation course. You find some subtours on some clustered subsets of the laser-point locations and then patch these subtours together into a tour

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{1000}, y_{1000})$$

of all the laser-points. Finding that the total length is 4.99999 metres, you think 'Aaaah, redundancy is staved off another week'. Now you just have to prove to the boss that the answer to the question *Is there a tour of length at most* 5 *meters?* is yes. But this is trivial (even for the boss). You just hand in your tour and then he/she simply checks the following inequality:

$$\sqrt{(x_{1000} - x_1)^2 + (y_{1000} - y_1)^2} + \sum_{i=1}^{999} \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} \leq 5$$

This last calculation can be done trivially by a computer with $n + 1$ (here, $n = 1000$) basic operations, i.e., $n$ additions $+$ and 1 comparison $\leq$. The point being that the boss has no idea that you did not check all 999! possible tours, only that you found one that worked and that you could prove quickly that it did work.

Let's dispense now with the bedtime stories and give a formal definition.

**Definition:** A decision problem $D(L)$ is in $\mathcal{NP}$ if there is a polytime algorithm $M$ (called the *checking algorithm*), and polynomial $p$, which have the following properties:

- $\forall x \in L$ : there is a *certificate* $C_x \in \Sigma^*$ such that:
  (i) $size(C_x) \leq p(n)$, where $n := size(x)$
  (ii) $M$ accepts the augmented input $(C_x, x)$,

- $\forall x \notin L$ : no input $(C, x)$ is accepted by $M$.

The formal definition says that a language $L$ is in $\mathcal{NP}$ if there is a polytime algorithm which given $x \in L$ and some 'extra information', called the certificate, can verify that $x$ is indeed in $L$. It should be clear from this definition that $\mathcal{P} \subseteq \mathcal{NP}$ since if $A$ is a polytime algorithm which solves the decision problem for $L$, then it will also suit as the checking algorithm in the definition of $\mathcal{NP}$. We do not even need to give $A$ any extra information or certificate, it already leads to state $q_Y$ on the input $x$.

The term 'nondeterminism' emanates from the fact that the certificate is somehow guessed and not computed. In other words, it is only required that there **exists** some certificate which when given to the checking algorithm, will lead to $q_Y$. How can this be used to devise a practical (i.e., deterministic) algorithm to solve a problem? Tantalizingly, this remains one of the most important open problems in mathematics:

**Conjecture 6.1** $\mathcal{P} \neq \mathcal{NP}$.

This question has perplexed many researchers and in fact the best that can (currently) be proved is the following:

**Exercise 6.1** *If $\Pi \in \mathcal{NP}$, then there is a polynomial $p$ such that $\Pi$ is solved by some algorithm with running time $O(2^{p(n)})$.*

This is a long shot from giving a polytime algorithm for $\Pi$. In fact, it would not be going against common belief to think that being allowed to guess (or have Merlin on your side) is somehow a very great advantage. One would surely expect that there are some decision problems which can be solved using nondeterminism in polynomial time, but which do not actually have (deterministic) polytime algorithms.

### Nondeterministic Turing Machines

A *nondeterministic Turing machine* (NDTM) is the same as a Turing machine except that at each step it may guess from a number of possibilities. Thus a transition function of a program for an NDTM now

specifies a non-empty set of "possible next steps" of the computation and so is of the form:

$$f : (Q - \{q_Y, q_N\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, -\}) - \emptyset.$$

A *computation* of such a program by the NDTM consists of an execution where at each step, the machine (in state $q$, reading symbol $\sigma$) performs a step specified by one of the triples in the set $f(q, \sigma)$. Thus there are, in general, many different computations for each fixed input $x$.

A NDTM program *solves* the decision problem $D(L)$ if (i) whenever $x \in L$, **some** computation terminates in the state $q_Y$ and (ii) whenever $x \notin L$, no computation halts in state $q_Y$. Note that in case (ii) we do not require the NDTM to halt.

The *time complexity function*, denote by $T_A(n)$, of a nondeterministic algorithm (i.e., program) is

$$T_A(n) = \max\{\{1\} \cup \{m : \exists x \in \Sigma^* \text{ of length } n \text{ such that there is some computation of } A \text{ on input } x \text{ which terminates after } m \text{ steps}\}\}.$$

Thus by convention, if $A$ does not halt for any input of length $n$, then $T_A(n)$ is defined to be 1. The above concepts give now an alternative definition of $\mathcal{NP}$.

**Theorem 1** *A problem $D(L) \in \mathcal{NP}$ if and only if there is a polytime nondeterministic algorithm which solves $D(L)$.*

**Sketch of Proof:** ($\Rightarrow$) Suppose first that $D(L) \in \mathcal{NP}$ and so there exists a polytime algorithm $A$ and polynomial $p$ as given in the definition. Consider nondeterministic algorithm $B$ which does the following:

1. Writes some string $C$ of length at most $p(|x|)$ beside the input string $x$,
2. Simulates $A$ on input $(C, x)$.

Phase 1. is nondeterministic since at each step it **chooses** one of the symbols from $\Gamma$ to write on the tape and then moves left. If $x \in L$,

then there is clearly a computation of $B$ which ends in $q_Y$, namely the computation where the string $C_x$ is written on the tape in Phase 1. Conversely, if $x \notin L$, then by the definition of $A$, no matter what string is written in Phase 1, the computation will not halt in state $q_Y$. Thus $B$ is a nondeterministic algorithm which solves $D(L)$.

To see that the algorithm $B$ is polytime, note that its running time is bounded by number steps in Phase 1. added to the number of steps in Phase 2. If the running time of $A$ is bounded by a polynomial $Cn^k$, then since the string $C$ has length at most $p(n)$, $B$'s running time is at most $p(n) + C(n + p(n))^k$. Since $p(n)^k$ is again a polynomial, $B$ has polynomially bounded running time.

**Exercise 6.2** *Prove the converse is true.*

□

We have already noted the asymmetry of the definition of $\mathcal{NP}$ as it only requires there to be a quick verification of Yes-ness (and not No-ness). This prompts us to define $L^c = \{x \in \Sigma^* : x \notin L\}$ and $co\mathcal{NP} = \{D(L) : D(L^c) \in \mathcal{NP}\}$. Diagrammatically, this looks like:

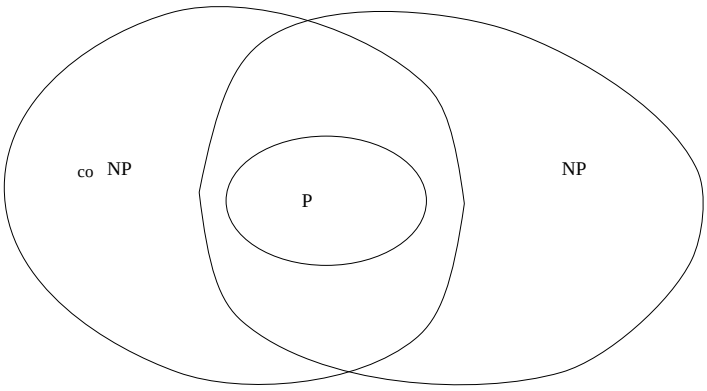


Figure 5:

It is easily deduced that any problem in $\mathcal{P}$ is also in $\mathcal{NP}$ and $co\mathcal{NP}$. The converse of this is also widely believed although perhaps not as widely as is Conjecture 6.1.

**Conjecture 6.2** $\mathcal{P} = \mathcal{NP} \cap co\mathcal{NP}$.

**Solutions:**

**Exercise 6.2.** Suppose that $\Pi$ is a problem which is solved by a polytime nondeterministic algorithm $M = (\Sigma, Q, \delta)$. Now consider an algorithm $A = (\Sigma, Q \times Q', \delta')$ which takes as input a string $C, x$ where $x$ is written on the squares $1, 2, \ldots$, called the *working part* of the tape and $C$ is written on the squares $0, -1, -2, \ldots$, called the *read-only part*. $A$ will simulate a certain computation of $M$ on input $x$. The specific computation which $A$ simulates is determined by the string $C$. Thus $C$ is a string which is the encoding of a list of triples: $\ldots (q_3, \sigma_3, m_3), (q_2, \sigma_2, m_2), (q_1, \sigma_1, m_1)$ (where each $q \in Q, m_i \in \{L, R, -\}$). Each $(q_i, \sigma_i, m_i)$ may be used to indicate step $i$ of a computation of the nondeterministic algorithm $M$. $A$ tries to perform these steps $1, 2, 3, \ldots$ of $M$ on the input string $x$.

The extra states $Q'$ are needed (i) to 'decipher' the strings representing these steps into the correct move (and to make sure that the encoding is correct!, i.e., that it is really a string which represents some state $q$ and move $L, R, -$) and (ii) to be able to repeatedly move the tape head back and forth between the working part the read only parts of the tape. Thus $A$ has, if you will, two subroutines which it repeatedly calls. During execution of these subroutines $A$ will never change the first component of its state, i.e., it will move from state $(q, q')$ to $(q, q'')$ etc.. Roughly, $A$ has then the following form:

Repeat until $A$ enters some state $q, q'$ where $q \in \{q_Y, q_N\}$.

**Phase 1.** Subroutine Decipher.

**Update First State Component**

**Phase 2.** Subroutine Tape Manipulate.

At step $i$, if $A$ is in state $(q_{i-1}, q')$ and the last symbol it scanned on the working part of the tape was $\sigma_{i-1}$, then it enters Phase 1. to check whether $q_i, \sigma_i, m_i$ is a valid triple. During this phase it checks whether $C$'s encoded triple $(q_i, \sigma_i, m_i)$ is in fact a possible step for $M$, i.e., it checks if it is in the set $\delta(q_{i-1}, \sigma_{i-1})$. Suppose that it completes this phase in state If not then it enters state $(q_N, q')$ and otherwise it enters state $(q_i, q')$. It then proceeds to Phase 2. (which again only changes

the second state component) which tells the tape head to move to the current square of the working tape and write the symbol $\sigma_i$, and then to move the head according to $m_i$ (and record the resulting next current square of the working tape). The tape head then moves back to the read only part and repeats Phase 1.

**Appendix:** *Provably Intractible Problems*

## 7 NP-completeness

What does it mean for a problem $A$ to be at least as hard as a problem $B$? A natural interpretation would be that if you could solve $A$, then you could also solve $B$. Keeping this in mind, we define the following:

**Definition 7.1** *A polynomial transformation of a decision problem $D(L_1)$ to a decision problem $D(L_2)$ is a function $f : \Sigma^* \to \Sigma^*$ such that*

- $\exists$ *polytime algorithm which computes $f$*
- *for all $x \in \Sigma^*$, $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

If there is such a polynomial transformation, then we say '$D(L_1)$ transforms to $D(L_2)$' and denote this by $D(L_1) \propto D(L_2)$.

**Theorem 2** *If $D(L_1) \propto D(L_2)$ and $D(L_2) \in \mathcal{P}$, then $D(L_1) \in \mathcal{P}$.*

**Proof:** Let $A$ be a polytime algorithm which solves $D(L_2)$ and $B$ be a polytime algorithm which computes a polynomial transformation $f$ from $D(L_1)$ to $D(L_2)$. Then the following is a polytime algorithm $C$ which solves $D(L_1)$.

1. Given input $x$, use $B$ to compute $f(x)$
2. Apply $A$ to the output $f(x)$ of $B$.

Note first that the number of steps taken by $C$ on input $x$ is at most

$$T_B(|x|) + T_A(|f(x)|). \tag{3}$$

Since $B$ is polytime, its running time is bounded by $C_1 n^{k_1}$ for some $C_1, k_1 \geq 1$. Hence clearly it could not write a string longer than this on the tape. Thus $|f(x)| \leq C_1(|x|)^{k_1}$. Plugging this into (3) we see that if $A$ has running time bounded by $C_2 n^{k_2}$, then $T_C(n) \leq C_1 n^{k_1} + C_2(C_1 n^{k_1})^{k_2}$ and so it follows that $C$ has running time of order $O(n^{k_1 k_2})$.

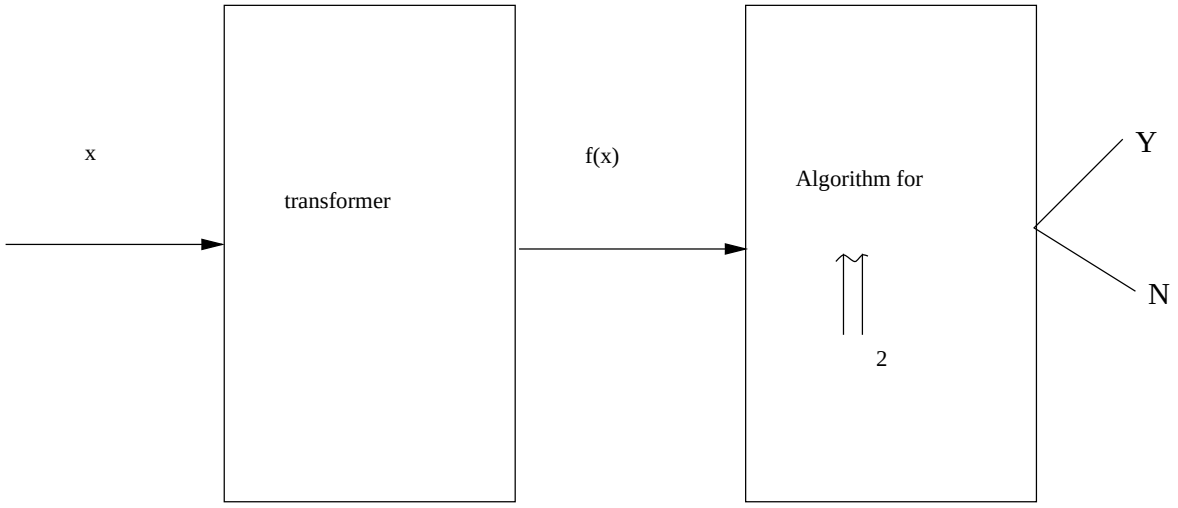


Figure 6:

It is easy to see that $C$ solves $D(L_1)$ since $C$ ends in state $q_Y$ on input $x$ if and only if $f(x) \in L_2$ if and only if $x \in L_1$ (by definition of $f$). Moreover, if $x \notin L_1$, then $f(x) \notin L_2$ and so $A$ halts in state $q_N$. □

Similar arguments to the previous proof also show the transitivity of the relation $\propto$.

**Theorem 3** *If* $\Pi_1, \Pi_2, \Pi_3 \in \mathcal{NP}$ *and* $\Pi_1 \propto \Pi_2$ *and* $\Pi_2 \propto \Pi_3$, *then* $\Pi_1 \propto \Pi_3$.

**Exercise 7.1** *Prove Theorem 3.*

We now capture what it means to be a "hardest" problem in $\mathcal{NP}$.

**Definition 7.2** *A problem* $\Pi \in \mathcal{NP}$ *is* NP-complete *if* $\Pi' \propto \Pi$ *for every other* $\Pi' \in \mathcal{NP}$.

It seems a daunting task to show the existence of an $\mathcal{NP}$-complete problem $\Pi$ since one would have to show one polynomial transformation from each other problem in $\mathcal{NP}$ to $\Pi$. This is, however, exactly what a landmark theorem of Cook asserts: *there exists an* $\mathcal{NP}$*-complete problem.*

We introduce his $\mathcal{NP}$-complete problem in the next section and the proof of its "completeness" in the section after that, but for now we note only that the task of proving problems $\mathcal{NP}$-complete becomes considerably easier once we know of a single $\mathcal{NP}$-complete problem.

**Theorem 4** *If* $\Pi$ *is* $\mathcal{NP}$*-complete, and* $\Pi^* \in \mathcal{NP}$ *such that* $\Pi \propto \Pi^*$, *then* $\Pi^*$ *is also* $\mathcal{NP}$*-complete.*

**Proof:** Choose any $L' \in \mathcal{NP}$. Since $L$ is an $\mathcal{NP}$-complete problem, $L' \propto L$ and so by Theorem 3 and the fact that $L \propto L^*$, we have $L' \propto L^*$. Since the choice of $L'$ was arbitrary, it follows that $L^*$ is $\mathcal{NP}$-complete . □

Note that Theorem 2 implies that if any $\mathcal{NP}$-complete problem is polynomially solvable, then every problem in $\mathcal{NP}$ is in $\mathcal{P}$ , thus refuting Conjecture 6.1.

**Theorem 5** *If* $\Pi$ *is* $\mathcal{NP}$*-complete and* $\Pi \in \mathcal{P}$ *, then* $\mathcal{P} = \mathcal{NP}$ *.*

On the other hand, to verify the conjecture, one must show that there is some $\mathcal{NP}$-complete problem $\Pi$ such that any algorithm solving $\Pi$ does not have polynomial running time. Finding such lower bounds is generally a difficult task, nevertheless in Appendix **??**, two examples of 'provably intractable' problems (outside the class $\mathcal{NP}$ ) are given.

# Notes for Nov 12, 1998

## 8 Decision Problems, Encoding and Satisfiability

Up to this point, a decision problem has been explicitly associated with a language $L$ over an alphabet $\Sigma^*$. We will require an adaptation of our terminology in order to speak about more practical problems in an unwieldy manner. To this end, we will consider a *decision problem* $\Pi$ to consist of a set of *instances* $D_\Pi$ and a subset $Y_\Pi$ of $D_\Pi$ called the *yes-instances*. We will normally specify such a decision problem by (i) the definition of a *generic instance* of the problem, which is given in terms of basic descriptors such as numbers, graphs, sets etc. and (ii) a Yes-No question. As an example we consider the following decision problem:

**Problem:** PRIME

**Instance:** An integer $k \geq 1$

**Question:** Is $k$ prime?

The set of instances $D_{Prime}$ of this problem are $\{1, 2, 3, ...\}$ and the Yes-instances are the prime numbers $\{2, 3, 5, 7, ...\}$. Note that any instance is obtained by plugging in a particular value for the descriptor $k$.

Let's slowly introduce a specification of our first $\mathcal{NP}$-complete problem in order to make this more concrete.

A *truth assignment* for a set $U = \{u_1, u_2, \ldots u_m\}$ of boolean variable is a function $t : U \to \{T, F\}$. If $t(u_i) = T$, then the variable is said to be *true under* $t$. For each variable $u \in U$, we associate two *literals* $u$ and $\overline{u}$. We say that the literal $u$ is *true under* $t$ if the variable $u$ is. And the literal $\overline{u}$ is *true under* $t$ if the variable $u$ is not true under $t$.

A *clause over* $U$ is a subset of the literals, e.g. $\{u_1, \overline{u}_9, u_{11}\}$. A clause is *satisfied* by the truth assignment if and only if at least one of its members is true.

A collection $\mathcal{C}$ of clauses is satisfiable if there is some truth assignment which satisfies all clauses in $\mathcal{C}$ simultaneously. We can now specify the first $\mathcal{NP}$-complete problem which is called SATISFIABILITY or SAT:

**Problem:** SAT

**Instance:** A finite set of boolean variables $U$ and a finite collection $\mathcal{C}$ of clauses over $U$.

**Question:** Is $\mathcal{C}$ satisfiable?

The above **Instance** section is the generic instance of the problem and a particular instance (i.e., an element of $D_{SAT}$ is obtained by giving specific values for each descriptor in the generic instance: e.g., $U = \{u_1\}$ and $\mathcal{C} = \{\{u_1\}, \{\overline{u}_1\}\}$ - note for this instance, the answer to the question is trivially NO.

Thus we speak of finding algorithms to solve such decision problems and this should mean that if the algorithm is given an input 'specifying a particular instance' $I$ of the problem, then it should output the correct answer Yes-No for $I$. The only problem is that our algorithms have only been equipped to take strings as inputs and to solve decision problems for languages. The solution lies in using some encoding scheme. For example, for the problem SAT, we could use a Turing machine with a tape alphabet $\Gamma = \{u, 0, 1, \ldots, 9, \{, \}, \neg\}$ and initially write down on the tape $\ldots\#\#u1\{u1\}\{\neg u1\}\#\#\ldots$, where $\neg u1$ will be understood by the algorithm to mean the literal $\overline{u}_1$. If the algorithm can be made to 'understand' this, then surely we could just as well use some string 01010001 to be 'understood' as $\{$, and another string to mean $u$ etc. In short, we can make an unambiguous *string-encoding* of all possible values to be plugged in for parameters of the generic instance. Of course, the input would be a little longer, but only by a constant amount. For example, our original alphabet had 14 symbols and under our encoding, each of them will be assigned some string in $\Sigma^*$. If $M$ is the longest of these 14 strings, then input under our new encoding will be at most $M$ times the length of the original input. This is peanuts to us since we are interested in algorithms with polynomial running time and these are impervious to constant factors.

The one point to be careful about is that we do require the encoding to be 'reasonable' in the sense that (i) it should be compact and not padded by extra blanks (this is cheating and could allow our algorithm to take many extra steps as a function of the input length) and (ii) numbers are represented by binary notation or by decimal notation or by octal notation but **not** by monadic/unary notation.

Now suppose that we have fixed some reasonable encoding $e$ for a decision problem $\Pi$. If we say that $\Pi$ belongs to $\mathcal{NP}$ or $\mathcal{P}$ etc., we will implicitly be referring to the decision problem for the language:

$$L[\Pi, e] = \{x \in \Sigma^* : \Sigma \text{ is the alphabet used by } e \text{ and } x \text{ is the encoding of an instance } I \in Y_\Pi\}$$

Thus at the heart of it all, we are still referring to the acceptance of strings in a language but to avoid tedium, we stay as far as possible away from making specific which encoding we use. This is because under our assumptions, two reasonable encodings $e_1, e_2$ will be *polynomially equivalent* in the sense that there exists a polytime algorithm to solve $D(L[\Pi, e_1])$ if and only if there is one to solve $D(L[\Pi, e_2])$.

## 9 Cook's Theorem

**Theorem 6** SAT *is* $\mathcal{NP}$*-complete .*

**Proof:** Clearly SAT $\in$NP since a satisfying truth assignment is a certificate which is checked by an easy polytime algorithm. It remains to show that for each $\Pi = D(L) \in$NP , $\Pi \propto$ SAT.

**Outline:** The idea of the proof is to describe a generic construction of a polynomial transformation $f_\Pi$ for any $\Pi \in$NP . This is done as follows. Since $\Pi \in$NP , there is a polytime nondeterministic algorithm $M$ which solves it. Suppose $M$ has states $\{q_0, q_1, \ldots, q_{r-1}, q_r\}$ where $q_{r-1}, q_r$ represent the states $q_N, q_Y$ respectively. Then for each $q_i$ and each symbol $l \in \{0, 1, \#\}$, the transition function gives a set $\delta_M(q_i, l)$ of possible steps. We first claim that we may choose $M$ so that each of these sets has size (at most) two (Exercise). Thus each entry of $M$'s program table consists of a set of size two and for each entry we arbitrarily label the members of the set as *left* and *right*. We assume now that such a program is fixed for $\Pi$ and that we are given a polynomial $p(n)$ such that $T_M(n)$, the time complexity function of $M$, satisfies $T_M(n) \leq p(n)$ for each $n$. For simplicity, in fact, assume that $T_M(n)$ is itself such a polynomial.

We now describe a polytime algorithm which given an $x \in \Sigma^*$ constructs an instance $F_M(x)$ of SAT (i.e., a set of clauses) such that $F_M(x)$ has a satisfying truth assignment if and only if $x \in L$, or in other words, $F_M(x)$ is satisfiable if and only if $M$ has an accepting computation of length at most $T_M(|x|)$ on input $x$. [1]

We now show how to construct the instance $F_M(x)$ of SAT. This instance has the following four classes of variables. One should presently think of the right hand column below as indicating how we will want these variables to behave in any satisfying truth assignment.

[1] Note that this does not explicitly define a polynomial transformation $f_\Pi$ but rather a map from each string $x \in \Sigma^*$ to an instance of SAT. However, if $g_e$ is the mapping of any reasonable encoding of SAT, then $f_\Pi = g_e \circ F$ yields the desired transformation.

| Variables | Interpretation |
|---|---|
| $Q[i,k] : i = 0,1,\ldots T_M(n); k = 0,\ldots,r$ | True if the machine is in state $q_k$ at time $i$ |
| $H[i,j] : i = 0,1,\ldots T_M(n); j = -T_M(n),\ldots,$ $-1,0,1,\ldots,T_M(n)$ | True if the tape head is scanning square $j$ at time $i$ |
| $S[i,j,l] : i = 0,1,\ldots T_M(n); j = -T_M(n),\ldots,$ $-1,0,1,\ldots,T_M(n), l = 0,1,\#$ | True if square $j$ contains symbol $l$ at time $i$ |
| $L[i] : i = 0,1,\ldots T_M(n),$ | True if we execute the 'left' step at at time $i$ |

Note that the number of variables of the instance $F_M(x)$ is bounded by $(T+1)(r+1)+(T+1)(2T+1)(1+3)+(T+1) \leq (T+1)(8T+6+r)$ where $T = T_M(n)$ and $n = |x|$. Thus since $r$ is fixed and $T_M(n)$ is bounded by a polynomial, the number of variables is polynomially bounded.

We now describe the clauses of $F_M(x)$. The idea is to add several families of clauses which force our variables to behave as we wish. That is, they represent an accepting computation of $M$ on the input string $x$. These families enforce the following properties of any nondeterministic TM computation:

(a) The machine is in exactly one state at any time.

(b) The machine is scanning exactly one square at any time.

(c) Each tape square contains exactly one symbol at any time.

(d) Computation is started in state $q_0$ scanning square 1 and the tape properly contains the string $x$.

(e) Computation finishes in $q_r$.

(f) Computation obeys $M$'s transition function:

(f1) If machine is not scanning square $j$ at time $i$, then the content of square $i$ is the same at step $i+1$.

(f2) If square $j$ contains $l$ and the machine is scanning square $j$ at time $i$ in state $q_t$. Then at time $i+1$, the contents of square $j$ is $\delta_M(q_t, l)$.

We now describe the clauses which enforce the above rules. The family (a) can be split into two types of clauses.

(a1) At each time $i$, the machine is in **at least** one state.

These are achieved by the following clause for each $i$:

$$Q[i,0] \vee Q[i,1] \vee \ldots \vee Q[i,r]$$

Then we also have clauses to enforce that

(a2) At each time $i$, the machine is in **at most** one state.

These are achieved by the following clauses:

$$\overline{Q[i,j]} \vee \overline{Q[i,j']}$$

for each $0 \leq j < j' \leq r$. This says that either the machine is **not** in state $j$ or **not** in state $j'$, i.e., it is note in both states at time $i$. The construction of the clauses for (b),(c) is also very similar. The clauses for (d) are trivial. They consist of one-element clauses $Q[1,0]$ and $H[1,1]$ which require the machine to start in state $q_0$ scanning square 1. We also force the correct initial contents of the tape (where $x$ is the string $\sigma_1 \ldots \sigma_n$) by way of the following single element clauses: $\{S[1,1,\sigma_1]\}, \{S[1,2,\sigma_2]\}, \ldots, \{S[1,n,\sigma_n]\}$ and then $\{S[1,j,\#]\}$ for each $j = -T, \ldots, 0$ and $j = n+1, n+2, \ldots T$. Finally, (e) says that at some point in time, state $q_r$ is reached: $Q[0,r] \vee Q[1,r] \vee \ldots \vee Q[T,r]$.

We achieve (f1) with the following clauses:

$$\overline{S[i,j,l]} \vee H[i,j] \vee S[i+1,j,l]$$

which can be parsed as: *if $H[i,j]$ is false (we are not scanning square $j$ at time $i$)* **and** $\overline{S[i,j,l]}$ *is false (square $j$ contains $l$ at time $i$), then $S[i+1,j,l]$ must be true (square $j$ still contains $l$ at time $i+1$).*

Now we must enforce property (f2).

For each $q_k \in Q$, and $l \in \{0,1,\#\}$ let $\delta_M(q_k,l)$ consist of a left move $(q_{k'}, l', \Delta')$ and a right move $(q_{k''}, l'', \Delta'')$.[2] In particular, if $L[i]$ is true

[2]Here, we use $\Delta = -1, 0, 1$ to represent moves $L, -, R$ respectively.

then we must obey the first action, otherwise we obey the second. Thus we add the constraints

$L[i] \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee H[i+1, j+\Delta'']$
$L[i] \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee Q[i+1, k'']$
$L[i] \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee S[i+1, l'']$

to ensure that we exceute the right move at time $i$ if $L[i]$ is false. Similarly we add

$\overline{L[i]} \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee H[i+1, j+\Delta']$
$\overline{L[i]} \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee Q[i+1, k']$
$\overline{L[i]} \vee \overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee S[i+1, l']$

to ensure that if $L[i]$ is true, then at time $i$ we exceute the left move specified by $M$'s transition function. As for (f1), the first of the above clauses can be parsed as: *if the first three literals are false, then at time $i$ we are (i) scanning square $j$ and this square contains symbol $l$, (ii) we are in state $k$ and (iii) we are supposed to execute the left move, then the last literal must be true in any satisfying truth assignment. That is, the head position at time $i+1$ must be $j+\Delta'$.*

That is the complete construction of $F_M(x)$ and it is easily seen that these clauses can be written down in polynomially bounded time.

Now if $x \in L$, then $M$ has an accepting computation on input $x$ which takes at most $T_M(n)$ steps. Thus we can find a truth assignment which makes all of the clauses from (a)–(f) true simultaneously. (Convince yourself of this.)

On the other hand, if $F_M(x)$ has a satisfying truth assignment, then the construction guarantees that it corresponds to some accepting computation of $M$ under the interpretations of the variables given at the outset, and so $x \in L$. Thus $x \in L \Leftrightarrow F_M(x)$ is satisfiable and we are finished. □

**Exercises**

**[1]** Show that if $\Pi \in \mathcal{NP}$, then it is solved by some nondeterministic polytime algorithm $M$ such that $|\delta_M(q,l)| \leq 2$ for each state $q$ of $M$ and symbol $l$.

## 10 Proving NP-completeness Results

### 10.1 The 3-SAT, Vertex Cover, Stable Set and Clique Problems are $\mathcal{NP}$-complete

We next show that the following problem is $\mathcal{NP}$-complete : **Problem:** 3-SAT

**Instance:** An instance of SAT for which each clause has at most three (or exactly three) literals.

**Question:** Is the collection of clauses satisfiable?

It is easily checked that 3-SAT $\in$NP and so as we saw earlier, it will be enough to show that SAT polynomially transforms to 3-SAT. We do this by transforming an instance $I$ of SAT into an instance $I'$ of 3-SAT by introducing new variables so that we may eliminate clauses with four or more literals. Suppose for instance that $C = \{z_1, z_2, \ldots, z_k\}$, $k \geq 4$, is a clause appearing in the instance $I$. For any such clause we introduce $k-3$ new variables $y_1, y_2, \ldots, y_{k-3}$ associated with $C$. We then replace the clause $C$ in $I$ with the following clauses which will appear in $I'$:

$$\{z_1, z_2, y_1\},\ \{\overline{y_1}, z_3, y_2\},\ \{\overline{y_2}, z_4, y_3\}, \ldots, \{\overline{y_{k-4}}, z_{k-2}, y_{k-3}\}, \{\overline{y_{k-3}}, z_{k-1}, z_k\}.$$

Once we have done this for each 'large' clause of $I$, we claim that $I \in Y_{SAT}$ if and only if $I' \in Y_{3-SAT}$. To see this, suppose first that $t$ is a satisfying truth assignment for $I'$. We claim that the restriction of $t$ to the variables of $I$ is also satisfying for $I$. Clearly each clause of $I$ with at most three literals is satisfied. Suppose then that a clause $C = \{z_1, \ldots, z_k\}$, $k \geq 4$ is not satisfied. Then each literal $z_i$ is false under $t$. Since the expanded collection of clauses in $I'$ associated with $C$, are all satisfied we have that $y_1$ is true under $t$. But this implies that $y_2$ must also be true under $t$, which in turn implies that $y_3$ is true under $t$ and so on. This leads to the conclusion that $y_{k-3}$ is true under $t$ and hence the final clause $\{\overline{y_{k-3}}, z_{k-1}, z_k\}$ is not satisfied, a contradiction.

Conversely if $t$ is a satisfying truth assignment for $I$ then we can construct a satisfying truth assignment $t'$ for $I'$ as follows. The variables common to $I$ and $I'$ are given the same values under $t'$. Then for each large clause $C = \{z_1, \ldots, z_k\}$ of $I$, there is some $z_l$ which is evidently

true under $t$. Thus we make the variables $y_1, \ldots, y_{l-2}$ true under $t'$ and the remaining $y_i$'s are false under $t'$. This ensures that the expanded collection of clauses are all satisfied by $t'$.

Finally, we leave it as a routine exercise that the construction of the instance $I'$ can be created in polynomial time. Thus we have described a polynomial transformation from SAT to 3-SAT and hence 3-SAT is also $\mathcal{NP}$-complete .

We now show a transformation of 3-SAT to the vertex cover problem VC.

**Problem:** VC

**Input:** A graph $G$ and integer $k$.

**Question:** Does $G$ have a vertex cover of size $k$?

For those who have not studied graph theory, a *vertex cover* of a graph $G$ is a subset $S$ of the vertices such that each edge is incident to (or touches) some vertex in $S$. Once again it is trivial to show that VC $\in$NP and so we can show that it is $\mathcal{NP}$-complete by showing that 3-SAT is polynomially reducible to VC. We now describe such a transformation.

Let $I$ be an instance of 3-SAT with variables $U = \{u_1, \ldots, u_n\}$ and clauses collection $\mathcal{C} = \{c_1, \ldots, c_m\}$. We construct an instance $(G, k)$ of VC as follows. First we define $k = n + 2m$. Now the graph $G$ is defined in two parts. We first create *selector vertices* $u_1, \ldots, u_n, \overline{u_1}, \ldots, \overline{u_n}$. We also add an edge $u_i\overline{u_i}$ for each pair of selector vertices. Note that any vertex cover of $G$ will require at least one of $u_i, \overline{u_i}$ in order to cover this edge. This will correspond to whether we want to set the variable to true or false. Next, for each clause $c_i$ we create a triangle of vertices $v_i^1, v_i^2, v_i^3$. Note again, that any vertex cover in a graph containing such a triangles, must contain at least two of the $v_i^j$'s in order to cover the three edges of the triangle. We will still add more edges between the $m$-triangles and the $n$ edges, but note that the preceding comments have already shown that any vertex cover must contain at least $2m + n$ vertices, i.e., $k$ vertices. We will add the remaining edges in such a way that $G$ will have a vertex cover of exactly $k$ vertices if and only if $I$ was satisfiable.

Finally, if $c_i$ is a clause containing the literals, $z_1, z_2, z_3$ say, then we

join $v_i^1$ to the appropriate selector vertex $z_1$, $v_i^2$ to $z_2$ and $v_i^3$ to $z_3$.
**Exercise** Prove that $I$ is satisfiable if and only if $G$ has a vertex cover of size $2m + n$.

### 10.2 Hamilton Cycle is $\mathcal{NP}$-complete

A *Hamilton* cycle of a graph $G = (V, E)$ is a connected subgraph with edges $e_0, e_1, \ldots, e_n$ such that each vertex is incident to exactly two of the edges. The Hamilton Cycle Problem (HAM) is that of deciding whether a given graph contains a Hamilton cycle.

**Theorem 7** HAM *is* $\mathcal{NP}$*-complete .*

**Proof:** As before it is trivial to see that HAM $\in$NP since a certificate is simply some ordering $v_0, v_1, \ldots, v_{n-1}$ of the vertices and there is a simple checker algorithm which determines if for each $i = 0, 1, \ldots, n-1$, there is an edge joining $v_i$ and $v_{i+1}$ (modulo $n$).

To show that HAM is $\mathcal{NP}$-complete we polynomially transform Vertex Cover VC to HAM. Thus for each instance $I = (G, k)$ of VC we construct a graph $G_I$ which has a Hamilton cycle if and only if $G$ has a vertex cover of size $k$.

The graph $G_I$ will have two types of vertices: *selector vertices* $a_1, \ldots, a_k$ and *cover-checkers* $(x, e, 1), (x, e, 2), (x, e, 3), (x, e, 4), (x, e, 5), (x, e, 6)$ for each vertex of $G$ and each edge $e$ incident to $x$.

For each edge $e = xy$ of $G$ we construct a gadget amongst its cover-checkers as displayed in Figure 7.

This subgraph is called an *e-block* and the key point is that none of its internal vertices $(x, e, 2), (x, e, 3), (x, e, 4), (x, e, 5), (y, e, 2), (y, e, 3), (y, e, 4), (y, e, 5)$ are joined to any other vertex in the graph $G_I$. In particular, this implies that any Hamilton cycle of $G_I$ will have to pass through $G_I$ in one of the three ways drawn in Figure 8.

To see why this is true, suppose that $H$ is a Hamilton cycle. Note that since $(x, e, 2)$ has degree exactly two in $G_I$, any Hamilton cycle must

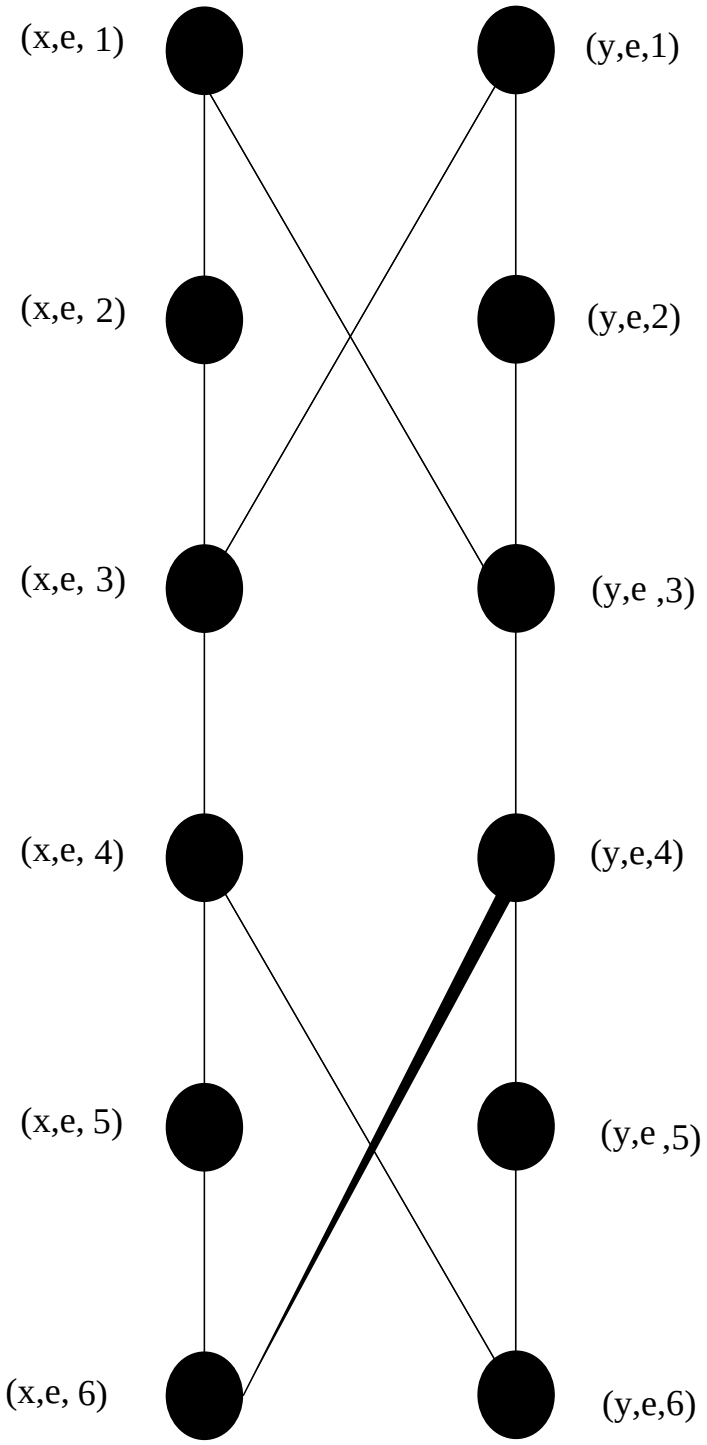


Figure 7: *Edge block for* $e = xy$.

use the edges $(x,e,1)(x,e,2)$ and $(x,e,2)(x,e,3)$. Similarly $H$ contains the edges $(y,e,1)(y,e,2)$, $(y,e,2)(y,e,3)$ and also the corresponding four edges at the other end of the $e$-block. Since any vertex has degree two in $H$, we also have that either (i) the edge $(x,e,3)(y,e,1)$ is in the cycle or (ii) the edge $(x,e,3)(x,e,4)$ is in the cycle. If Case (i) occurs, then it is easily deduced that the cycles passes through the $e$-block as in Figure 8(a). In Case (ii), one repeats the argument on the other side to deduce that $H$ passes through as in (b) or (c). The main point of all this is that if we are 'travelling' along $H$ and enter the $e$-block through $(x,e,1)$, then we exit through $(x,e,6)$ (and similarly for the $y$ side).

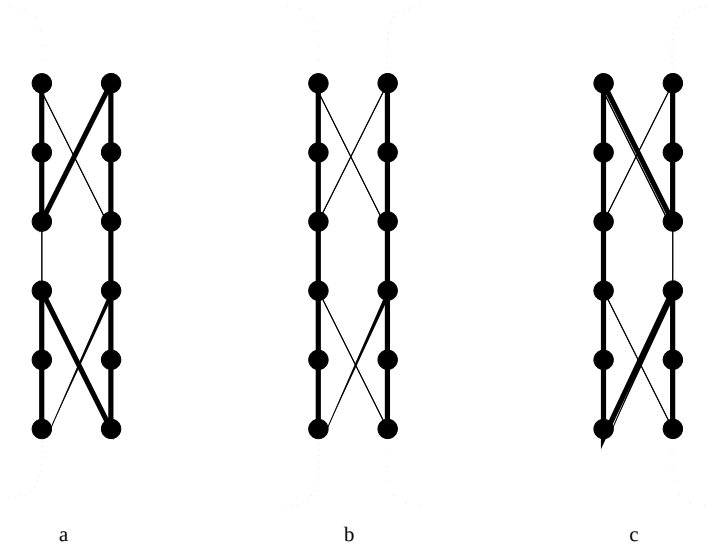


Figure 8:

We now add some more edges to the constructed graph $G_I$. For each $x$ in the old graph $G$, order its incident edges arbitrarily as $e_1, e_2, \ldots, e_d$. We link together these $e_i$-blocks as follows: for $i = 1, \ldots, d-1$, join $(x,e_i,6)$ to $(x,e_{i+1},1)$. This forms a *chain* (see Figure 9) $C_x$ of these $e_i$-blocks and we call $(x,e_1,1)$ and $(x,e_d,6)$ the *access points* of the chain. Note that for each $e = xy \in E(G)$, the corresponding $e$-block is in two chains of $G_I$, one for $x$ and one for $y$.

The final edges added to $G_I$ are joining each $a_i$ to the access points of $C_x$ for every $x \in V(G)$. We claim now that $G_I$ has a Hamilton cycle if an only if $G$ had a vertex cover of size $k$.

Suppose that $H$ is a Hamilton cycle of $G_I$. By renaming $a_i$'s we may assume that $H$ visits vertices in the order $a_1 P_1 a_2 P_2 \dots a_k P_k$ and then returns to $a_1$, where each $P_i$ consists of a tour through vertices in edge blocks only (i.e., $P_i$ does not contain selector vertices).

Now each $a_i$ is only adjacent to access points of chains so we may assume that the first vertex of $P_i$ is an access point lying on an $e$-block of some chain $C_{x_i}$, where $x_i \in V(G)$. We have seen that since $H$ enters this first $e$-block of $C_{x_i}$, on the $x_i$ side, then it also must exit the block on the $x_i$ side, but then by construction there is only one choice for the following vertex in the cycle, namely, to the beginning of another $e$-block in the same chain $C_{x_i}$. Thus $P_i$ must pass through the whole chain $C_{x_i}$, one block at a time, until it gets to the other endpoint. It then returns to vertex $a_{i+1}$. Note that we have not said precisely how $H$ traverses each $e$-block of $C_{x_i}$. It may traverse the block either as in Figure 8 (a) or (c).

Now that we understand the structure of a Hamilton cycle in $G_I$ we can show that the vertices $S = \{x_1, x_2, \dots, x_k\}$ (as just described) must be a vertex cover of $G$. This is because $H$ contains every vertex of $G_I$ and so for each edge $e = xy$ of $G$, the vertices $(x, e, j) \in G_I$ must be in $H$. We have argued that this is only possible if one of the chains $C_{x_i}$ contains the $e$-block. By construction, this only happens if $x_i$ is $x$ or $y$. Thus each $e$ is incident to some vertex of $S$ as desired.

Conversely we show that if the original graph $G$ has a vertex cover of size $k$, then $G_I$ has a Hamilton cycle. This is essentially the reverse of the preceding arguments. Suppose that $S = \{y_1, y_2, \dots, y_k\}$ is a vertex cover of $G$. Then $G_I$ has a Hamilton path which starts at $a_1$ and then visits an access point of $C_{y_1}$ and traverses (in some way that we describe in a moment) all of the blocks in this chain until reaching the other access point and then visits $a_2$, followed by $C_{y_2}$, $a_3$, $C_{y_3}$ etc. Now the

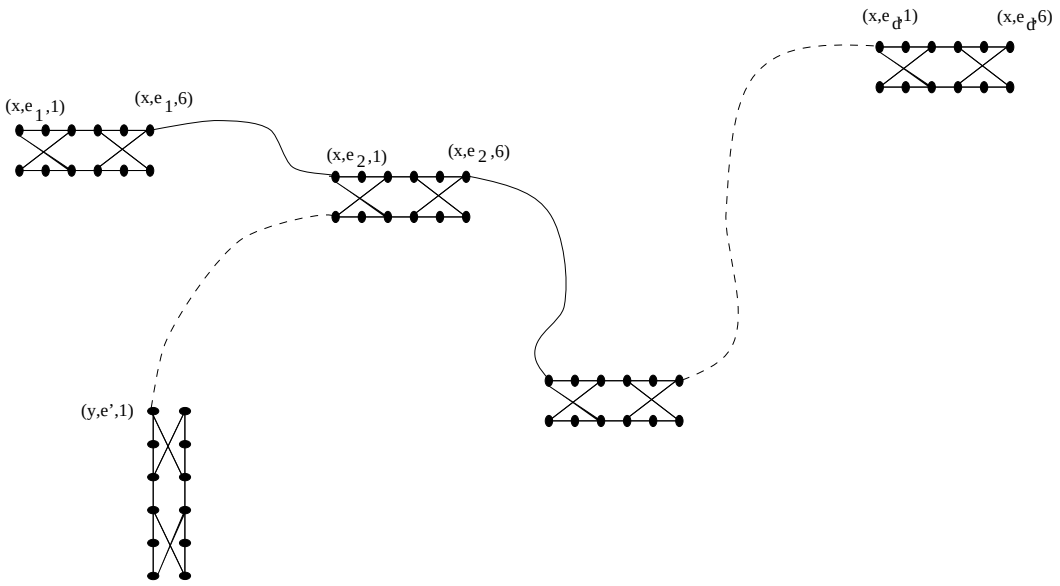


Figure 9: *A chain of edge blocks.*

way the tour visits the vertices of a specific edge block for an edge $xy$ is as follows: if both $x, y \in S$, then traverse the $xy$-block as in Figure 8(b). If one is in the set $S$, say $x$, then traverse as in Figure 8(a). Since $S$ is a vertex cover, at **least** one of $x, y$ is in $S$ and this is enough to guarantee that each edge block is visited and hence each vertex of $G_I$ is visited in this tour.

This completes the proof that $G$ has a vertex cover of size $k \Leftrightarrow G_I$ has a Hamilton cycle, and the theorem follows. □

### 10.3 Other $\mathcal{NP}$-complete Problems

**1:** Chromatic Number

**Instance:** Graph $G$, natural number $k \leq |V(G)|$.

**Question:** Is $G$ $k$-colourable?

*Comment:* Remains $\mathcal{NP}$-complete even for $k = 3$ and when the input is restricted to planar 3-regular graphs.

**2:** Planar Subgraph

**Instance:** Graph $G$, natural number $k \leq |V(G)|$.

**Question:** Does $G$ have an induced planar subgraph with $k$ vertices?

*Comment:* The related question - Is $G$ planar? - can be answered in polytime. Note that this is just the special case where $k = |V|$.

**3:** Subgraph Isomorphism

**Instance:** Graphs $G, H$.

**Question:** Does $G$ contain a subgraph isomorphic to $H$?

*Comment:* The related question: Is $G$ isomorphic to $H$? is not known to be in $\mathcal{P}$ or $\mathcal{NP}$-complete !

**4:** Max Cut

**Instance:** Graph $G$, integer $k$, weights $w(e)$ on the edges of $G$.

**Question:** Is there a proper nonempty subset $S$ of $V$ such that the sum of the weights of edges in the cut $\delta(S)$ is at least $k$?

*Comment:* The problem is not with sizes of the integers. It remains $\mathcal{NP}$-complete even when each $w(e) = 1$ and $k \leq |V|^2$.

**5:** 3 Dimensional Matching

**Instance:** Sets $V = V_1 \cup V_2 \cup V_3$ and $E \subseteq V_1 \times V_2 \times V_3$. each $|V_i| = m$.

**Question:** Is there a 3-D matching of size $m$? A matching is a subset $E'$ of $E$ such that no two elements of $E'$ agree in any coordinate.

**6:** Set Splitting = Hypergraph 2 Colouring

**Instance:** Collection $\mathcal{C}$ of subsets of a finite set $S$.

**Question:** Is there a partition of $S$ into two subsets $S_1, S_2$ such that no member of $\mathcal{C}$ is either in $S_1$ or $S_2$.

**7:** Subset Sum

**Instance:** A finite set $A$, weight function $w :\rightarrow \mathbf{N}$, integer $B$.

**Question:** Does there exists a set $A' \subseteq A$ such that $\sum_{a \in A'} w(a) = B$?

*Comment:* The comment is in $\mathcal{P}$ if we restrict to instances with bounded weights, e.g., if each $w(a) \leq |A|^{100}$. [3]

**8:** Bin Packing

**Instance:** Set of items $U$, size function $s : U \rightarrow \mathbf{N}$, bin capacity $B$, natural number $k$.

**Question:** Is there a partition of $U$ into subsets $U_1, U_2, \ldots, U_k$ such that the sum of the sizes in each $U_i$ is at most $B$?

[3] Such problems are said to be solvable in pseudo-polynomial time.

# Notes for Nov 26, 1998

## 11 Euclid's algorithm for greatest common divisors

The topic of this and the following sections are algorithmic problems concerning the factoring of integers. We will recall some facts from the elementary theory of numbers. All numbers considered here are integers. A positive integer $n$ that is the product of two numbers, $n = ab$, with $a > 1$ and $b > 1$, is called a *composite* number. A integer $n > 1$ that is not composite is called a *prime*. The problem

COMP. Input: A positive integer $n$.
Question: Is $n$ composite?

is one of the most natural problems in NP, since a certificate for compositeness is, for example, a divisor $a$ of $n$, so that it is easily checked that $n$ is divisible by $a$ without remainder. Note that the number $n$ is represented in binary, so the length of the input is the number of binary digits necessary to represent $n$, that is, $\lceil \log n \rceil$. (We assume in this section that all logarithms are binary, with basis two.)

On the other hand, it seems hard (a) to find factors of a number $n$, and (b) to provide a succinct certificate that $n$ is a prime, since in order to that, it looks as if one has to look at all integers up to $\sqrt{n}$ as potential factors of $n$. If $n$ is a 200-digit integer, then $\sqrt{n}$ has 100 digits, and dividing $n$ by something like $2^{100}$ integers takes very long. With $2^{\log n/2}$ operations, this is of course not a polynomial algorithm. There are much better factoring algorithms around, but they are still much away from being polynomial.

The complementary problem for COMP,

PRIME. Input: A positive integer $n$.
Question: Is $n$ prime?

is in coNP, but not obviously in NP. It is therefore a striking result, which we will prove in Section 13, that PRIME is in fact in NP. This means that (b) is solved: there is a polynomial-sized certificate that a number $n$ is prime. So both PRIME and COMP are in NP $\cap$ coNP.

On the other hand, both the certificate for COMP (a factorization) and the more complicated certicificate for PRIME, which we will learn, give no evidence about how to find a factorization of a large number $n$ quickly. That is, problem (a), factoring large integers, is still believed to be a computationally difficult problem, simply because no efficient factoring algorithm has yet been found, despite intense research efforts. A very interesting *cryptographic system*, the so-called RSA scheme for public key cryptography, is built on this belief that factoring is hard. Some people working in this area warn against this belief, since the encryption scheme would collapse once a fast factoring algorithm is found. After all, the problem COMP is both in NP and in coNP, so it may be just a matter of time until a polynomial factoring algorithm is found, in light of Conjecture 6.2 stating $\mathrm{P} = \mathrm{NP} \cap \mathrm{coNP}$. Others take the apparent difficulty of factoring as an indication that Conjecture 6.2 is false. The matter is unresolved at present.

Because of its interest in its own right, we will take the RSA scheme as our motivation to introduce the various number-theoretic concepts that we need. In this section, we first state some observations on divisibility.

Let $n$ be a positive integer. Computations *modulo* $n$ are performed as follows: Any integer $x$ has a unique representation of the form

$$x = qn + r, \qquad 0 \leq r < n$$

for some integers $q$ and $r$. In other words, $q$ is the quotient and $r$ the remainder when dividing $x$ by $n$. If the remainder is zero, then $x$ is also said to be *divisible* by $n$, written $n|x$, and $n$ is called a *divisor* of $x$.

If any two numbers $x$ and $y$ leave the same remainder when divided by $n$ (also called remainder *modulo* $n$), this is written $x = y (\mathrm{mod}\ n)$, to be read as "$x$ is equal to $y$ modulo $n$". A slightly more careful notation is $x \equiv y (\mathrm{mod}\ n)$, to be read as "$x$ is congruent $y$ modulo $n$", since having the same remainder is just an equivalence, not a an equality, but we will adopt the sloppier way of speaking. In any case, it is easy to see that $x = y (\mathrm{mod}\ n)$ holds if and only if $n|(x-y)$. For any $n$, the set of possible remainders modulo $n$ is denoted by $\mathbb{Z}_n$, so

$$\mathbb{Z}_n = \{0, 1, 2, \ldots n-1\}.$$

The notation $\mathbb{Z}_n$, similar to $\mathbb{Z}$ denoting the set of all integers, is used because one can add and multiply in $\mathbb{Z}_n$ just as one does with whole integers: just take the sum or product of two numbers $a$ and $b$ in $\mathbb{Z}_n$ as integers, and look at their respective remainder modulo $n$, to define their sum $a+b$ and product $ab$ in $\mathbb{Z}_n$. For example, let $n = 21$ and $a = b = 20$. Then $a + b = 40 = 19 (\text{mod } n)$, and $ab = 400 = 19 \cdot 21 + 1 = 1 (\text{mod } n)$. A simpler way to see the latter is to observe that $20 = -1 (\text{mod } n)$, hence $20^2 = (-1)^2 = 1 (\text{mod } n)$. When computing modulo $n$, it does not matter which integers are used to represent the respective remainder (this can be easily proved). Negative numbers, which can be useful for computation when they are smaller, can be used as well.

Given that one can add and multiply in $\mathbb{Z}_n$, a natural question is whether one can perform *division* as well. With respect to multiplication, $\mathbb{Z}_n$ has a neutral element 1, since $a\dot{1} = a$ and this equation holds also modulo $n$. Then the multiplicative inverse of a number $a$ in $\mathbb{Z}_n$ is an $x$ so that $ax = 1 (\text{mod } n)$. This equation cannot be solved for $x$ if $a$ is a number that has common divisors with $n$ other than one, since it means $ax + bn = 1$ for some integer $b$, and if $p|a$ and $p|n$ for some integer $p > 1$, then $p$ is also a divisor of $ax + bn$ which cannot be a divisor of 1.

This leads to the well-known concept of a *greatest common divisor* of two numbers $a$ and $b$, say, which is denoted by $gcd(a, b)$. By definition, this is the largest integer $k$ so that $k|a$ and $k|b$. Using the canonical factorisations of $a$ and $b$ into primes,

$$a = p_1^{a_1} p_2^{a_2} \cdots p_l^{a_l}, \qquad b = p_1^{b_1} p_2^{b_2} \cdots p_l^{b_l}$$

where $p_1, p_2, \ldots, p_l$ are the prime numbers that appear as factors of $a$ or $b$ and $a_1, \ldots, a_l, b_1, \ldots, b_l$ are their respective exponents (some of which may be zero if a prime is a divisor only of $a$ or $b$ alone), then it is easy to see that

$$gcd(a, b) = p_1^{\min(a_1, b_1)} p_2^{\min(a_2, b_2)} \cdots p_l^{\min(a_l, b_l)}.$$

However, this is not a good method for finding a gcd when $a$ and $n$ are large numbers since it requires factoring $a$ and $b$ first. The method of choice is the so-called *Euclidean algorithm*. Its input are two integers $a$ and $b$ and its output is $gcd(a, b)$. It works as follows:

1. Let $a = qb + r$ with integers $q$ and $r$ so that $0 \leq r < b$ (divide $a$ by $b$ with remainder $r$).
2. If $r = 0$, output $b$ as the gcd of the original numbers, stop.
3. If $r > 0$, go back to step 1 with $(a, b) := (b, r)$.

We illustrate this with an example: find $gcd(21, 27)$. Then we obtain the following equations (the first line indicating the general format):

$$\begin{aligned} a &= q \cdot b + r \\ 21 &= 0 \cdot 27 + 21 \\ 27 &= 1 \cdot 21 + 6 \\ 21 &= 3 \cdot 6 + 3 \\ 6 &= 2 \cdot 3 + 0 \end{aligned}$$

with output 3 for $gcd(21, 27)$. Note that the first line has $q = 0$ since $a < b$, so the first step amounts to a mere exchange of $a$ and $b$. It is avoided if one starts (without loss of generality, in fact) with $a > b$.

How many iterations are performed by the Euclidean algorithm? Given that $a > b$ (which is always the case after the first iteration since then $b > r$ and $b, r$ replace $a, b$ according to Step 3), we have $q \geq 1$ and therefore $2r < b + r \leq qb + r = a$. After any two iterations, $r$ has taken the place of $a$ and is less than half the size of $a$. Hence the computation terminates after at most $2 \log a$ iterations, so the computation time is polynomial in the input size since division with remainder can be performed in polynomial time.

The correctness of the Euclidean algorithm rests on the observation that any common divisor of $a$ and $b$ is also a divisor of $a - qb$, which is $r$, implying $gcd(a, b) = gcd(b, r)$. The output of the algorithm is, by construction, a number $k$ so that $gcd(a, b) = gcd(k, 0) = k$, which proves the correctness claim.

Furthermore, we can show the following property:

**Theorem 11.1.** *For any two positive integers $a$ and $b$, there are integers $x$ and $y$ so that $ax + by = gcd(a, b)$.*

*Proof.* The $m$ iterations of Euclid's algorithm can be written as (see the

above example) $b = b_1$,

$$\begin{aligned} a &= q_1 \cdot b_1 + b_2 \\ b_1 &= q_2 \cdot b_2 + b_3 \\ &\vdots \\ b_{m-2} &= q_{m-1} \cdot b_{m-1} + b_m \\ b_{m-1} &= q_m \cdot b_m + 0 \end{aligned}$$

where $b_m = gcd(a, b)$ is the output of the algorithm. These equations, omitting the last one, are equivalent to

$$\begin{aligned} a - q_1 \cdot b_1 &= b_2 \\ b_1 - q_2 \cdot b_2 &= b_3 \\ &\vdots \\ b_{m-2} - q_{m-1} \cdot b_{m-1} &= b_m \end{aligned}$$

which can be substituted backwards to obtain $b_m$ as a linear combination of the form $ax + by$. This is best illustrated with the above example,

$$\begin{aligned} 3 &= 21 - 3 \cdot 6 \\ &= 21 - 3 \cdot (27 - 1 \cdot 21) \end{aligned}$$

which is already of the desired form since the last equation that was used is $27 = 1 \cdot 21 + 6$. Here, 27 and 21 are the original numbers that we started with. □

With Theorem 11.1, we can easily observe which numbers in $\mathbb{Z}_n$ have multiplicative inverses: These are exactly the numbers $a$ with $gcd(a, n) = 1$, since then $ax + ny = 1$ holds for some integers $x$ and $y$, in other words, $ax = 1 (\text{mod } n)$. The multiplicative inverse $x$ in this equation is thus obtained as a by-product of the Euclidean algorithm with the backwards substitution explained in the proof of Theorem 11.1 (often called the "extended Euclidean algorithm").

If $gcd(a, n) = 1$, then $a$ is called *relatively prime* to $n$. The set of elements of $\mathbb{Z}_n$ that are relatively prime to $n$ and their number have the

notation

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid gcd(a,n) = 1\}, \qquad \Phi(n) = |\mathbb{Z}_n^*|.$$

By the preceding considerations, $\mathbb{Z}_n^*$ with multiplication is an abelian group. The group properties state that the operation is associative: $(xy)z = x(yz)$, has a unit: $x1 = x$, and for any $x$ an inverse $x^{-1}$ so that $x^{-1}x = 1$. It is abelian since $xy = yx$. These equations hold for all $x, y, z$ in the group. We can take a $k$-fold product of any element $x$ with itself, which is denoted $x^k$.

**Theorem 11.2 (Fermat).** *Let $gcd(x,n) = 1$. Then $x^{\Phi(n)} = 1 (mod\ n)$.*

*Proof.* Consider the set $S = \{xa \mid a \in \mathbb{Z}_n^*\}$. The elements $xa$ in $S$ are all distinct since $xa = xb$ implies $a = x^{-1}xa = x^{-1}xb = b$. Furthermore, they are all members of $\mathbb{Z}_n^*$ since $x \in \mathbb{Z}_n^*$. This proves $S = \mathbb{Z}_n^*$. Let $y$ be the product of all members in $\mathbb{Z}_n^*$, which is equal to $x^{\Phi(n)}y$ when considered as the product of all members of $S$. Multiplication with $y^{-1}$ shows $x^{\Phi(n)} = 1$ (in $\mathbb{Z}_n^*$) which is the statement of the theorem. □

We close this section with a simple case of determining $\Phi(n)$. Let $p$ and $q$ be two distinct prime numbers and $n = pq$. Then $\Phi(n) = (p-1)(q-1)$. Namely, any element in $\mathbb{Z}_n$ that is not relatively prime to $n$ is either a multiple of $p$ or of $q$. The former are the $q$ numbers $0, p, 2p, \ldots, (q-1)p$, the latter the $p$ numbers $0, q, 2q, \ldots, (p-1)q$, both of which we have to subtract from $n$. But then we have subtracted 0 twice, which is a multiple of both $p$ and $q$. All other elements of $\mathbb{Z}_n$ are in $\mathbb{Z}_n^*$, which proves $\Phi(n) = n - q - p + 1$ as claimed. After these preliminaries, we can explain the workings of a cryptographic system based on factoring.

## 12 Public Key Cryptography

Cryptography is a recent and rapidly expanding field, with many questions relating to computational complexity. The goal is to be able to send private messages between individuals, which cannot be intercepted by persons without proper authorization. The information necessary to

make a message private (encryption) and to decode it again is usually called a *key*. Like a physical latch key or the combination for a combination lock, there must be a sufficient number of keys around so that it is hopeless of trying them all. This is usually achieved by a sufficient number of bits representing this key, so that the number of possible keys is exponential in the key length.

One of the problems of secure communication is how to initiate it. Presumably, the parties could decide in advance on a certain cypher key, which they then use, but this is complicated and requires a lot of secrecy to start with. While useful in military environments, it is quite cumbersome for everyday exchanges. To circumvent this difficulty, the researchers Diffie and Hellman proposed in 1976 the concept of *public key cryptography*. Thereby, an individual (typically called "Alice") wishing to receive messages that only she can read, *publishes* an *encryption key* $E$ and a method how to send messages to her using that key. Any message $m$, say, is then converted to the encrypted message $m^E$ which is totally garbled, and can no longer be deciphered *even with the knowledge of* $E$, which is public anyhow. Only Alice has the suitable *decryption key* $D$ that, when applied to the cyphertext $m^E$ in the form $(m^E)^D$, gives back the original message $m$. Furthermore, the integrity of the message can also be checked since if the cyphertext $m^E$ is altered, any decoding of it will produce gibberish. In short, it is important that encryption and decryption are based on independent keys, which do not provide useful information about each other.

The advantages of a public key encryption scheme induced a search for a suitable method. In 1978, the researchers Rivest, Shamir, and Adleman (RSA) published the following scheme, which is widely regarded as a good solution to the problem.

Consider two large prime numbers $p$ and $q$, say with 200 binary digits each. We postpone the question of how to find such primes at the moment. (Basically, this works by taking a large number and subjecting it to a number of tests, each of which will detect that the number is *not* a prime with probability 1/2. Hence after 100 tests of this sort, a composite number will pass the verdict "yes, this is a prime" only with

probability $2^{-100}$ which is lower than the chance that other things go wrong, so the number will almost certainly be prime. Since prime numbers are sufficiently dense—about one in every $d$ numbers with $d$ digits is a prime—, one can thereby find a prime before long.)

Given the primes $p$ and $q$, take their product $n$, which has 400 digits in our example. Furthermore, consider a number $E$ that is relatively prime to $(p-1)(q-1)$. Note that $p-1$ and $q-1$ are composite numbers, so a bit of trying may be required to make sure that $E$ has no divisors in common with $p-1$ and $q-1$. Furthermore, $E$ should be a "random" number that gives no information about $(p-1)(q-1)$, so it should not be something like 1 or $-1$ modulo $(p-1)(q-1)$, for example. Then, find the multiplicative inverse $D$ of $E$ in $\mathbb{Z}^*_{(p-1)(q-1)}$ with the extended Euclidean algorithm, so that

$$DE = 1 (\text{mod } (p-1)(q-1)).$$

All this information is private to the receiver Alice of messages, who *publishes* $n$ and $E$. An encryption of a message then takes place as follows: Bob, who wants to sent a message that only Alice can read, breaks the message into pieces with 400 bits in length, so that each such piece is an integer $m$ with $0 < m < n$. It is not useful to have a message with $m = 0$. Then compute $m^E (\text{mod } n)$, which is another 400-bit integer between 0 and $n$. All this can be done based on the public information $E$ and $n$. Then the encrypted message piece is $m^E$, which is publically transmitted. The receiver Alice, and only she, then decrypts the message by computing $(m^E)^D$ modulo $n$, which for some integer $k$ is equal to

$$m^{ED} = m^{k(p-1)(q-1)+1} = (m^{\Phi(n)})^k \cdot m = 1^k \cdot m = m (\text{mod } n)$$

since $\Phi(n) = (p-1)(q-1)$ and by Fermat's Theorem 11.2. In fact, the latter reasoning applies only if the message $m$ and $n$ are relatively prime, but the above equation is also true if $m$ is a multiple of $p$ or $q$, as can easily be seen. In short, Alice has decrypted the message by exponentiation with $D$ modulo $n$.

In theory, Alice has no secrets at all, since anyone can intercept the message by first computing the factors $p$ and $q$ of $n$, and then inverting $E$

modulo $(p-1)(q-1)$ to obtain $D$. However, finding that factorization is presumably hard, whereas all these computations are easy once $p$ and $q$ are known. (Incidentally, the receiver Alice would be in deep trouble if anyone sent her a message that is a multiple of $p$ or $q$ since that person would have broken the code by finding the factorization of $n$; this concerns the use of Fermat's theorem mentioned in the preceding paragraph. We can just dismiss this case as unlikely.)

To illustrate a further use of this scheme, observe that the order of applying the keys can be reverted: Similar as above, $(m^D)^E = m^{DE} = m (\mathrm{mod}\ n)$. Here, $m^D$ is a message that only Alice can produce since only she knows $D$. Anyone can then decypher the message using $E$ and $n$ which are publically known to belong to Alice, so that the message $m$ has in fact Alice's *digital signature*. Something like this could be transmitted by Bob using his public key to authenticate himself. During this initial exchange, Alice and Bob can then also exchange a private key (e.g. randomly generated) based on more conventional encryption schemes which are more efficient to use.

This raises the question of the speed of the above computations. There, computing $m^E$ seems like an exponential procedure since $E$ has hundreds of digits so it is not clear how to multiply $m$ with itself $E$ times. The resulting number would surely be too big to make this a polynomial algorithm. However, all numbers are reduced modulo $n$ while the computation goes along, so they never exceed $n$ in length. Instead of multiplying $m$ with itself $E$ times, we repeatedly square it modulo $m$, giving the numbers $m^2, m^4, \ldots, m^{2^k}$ until $2^k$ exceeds $E$. Then with at most $k$ further multiplications modulo $n$ (as suggested by the binary representation of $E$), we obtain $m^E$ modulo $n$. The time required is a polynomial in $k$.

## 13 PRIME is in NP

see Papadimitriou, "Computational Complexity", Addison-Wesley, 1994, Sections 10.2 and 10.3.

So far we have discussed how to compare the relative difficulty of decision problems, but what about problems of the form:

**Problem.** Travelling Salesman Optimisation: (TSP-OPT)

**Instance:** An $n \times n$ matrix of pointwise (rational) distances $d_{ij}$.

**Find:** A tour (permutation of $\{1, \ldots, n\}$) $\pi_0 \pi_1 \ldots \pi_{n-1}$ such that $\sum_{i=0,\ldots n-1} d_{\pi_i \pi_{i+1}}$ is minimised.

In order to study such problems we introduce the notion of a *search problem* $\Pi$. This consists of a set $D_\Pi$ of instances and for each $I \in D_\Pi$ there is a set $S_\Pi[I]$ of *solutions*. An algorithm *solves* $\Pi$ if given an encoding of an instance $I \in D_\Pi$, it returns $\epsilon$ (the empty string) if $S_\Pi$ is empty and otherwise returns the encoding of some $s \in S_\Pi[I]$. Under a fixed encoding, a search problem will be identified with its *string relation* $R_\Pi$:

$$\{(x, y) \in (\Sigma^* - \{\epsilon\}) \times (\Sigma^* - \{\epsilon\}) : x \text{ is the encoding of an instance } I \text{ and } y \text{ is the encoding of some } s \in S_\Pi[I].\}$$

Clearly, with this terminology, any decision problem itself can be viewed as a search problem. (For example, $D(L)$ can be identified with the string relation $(x, \#)$ for each $x \in L$.)

For a string relation $R$, an *$R$-relativised oracle Turing machine* (or $R$-oracle TM) is the same as a 2-tape Turing machine except that if its *oracle tape* is active, then there is a *query* operation (in addition to $L, R, -$) which if the oracle tape contains the string $x$, then in the next step the oracle tape will contain (i) $\epsilon$ if there is no $y$ such that $(x, y) \in R$ and otherwise (ii) will contain $y$ such that $(x, y) \in R$. For example, in the case of the Hamilton cycle search problem, we could place the encoding $x$ of a graph $G$ on the oracle tape and in the next step, the tape will contain the encoding of some Hamilton cycle if $G$ has one.

**Definition 13.1** *For search problems $\Pi_1, \Pi_2$, $\Pi_1$ Turing reduces to $\Pi_2$, denoted by $\Pi_1 \propto_T \Pi_2$, if $\Pi_1$ is solved by a polytime $R_{\Pi_2}$-oracle Turing machine algorithm.*

We now have a new way of saying that one problem is at least as hard as another since if $\Pi_1 \propto_T \Pi_2$ and $\Pi_2 \in \mathcal{P}$ , then $\Pi_1 \in \mathcal{P}$ (convince yourself of this).

**Theorem 8** *If* $\Pi, \Pi'$ *are decision problems and* $\Pi \propto \Pi'$*, then* $\Pi \propto_T \Pi'$*.*

**Proof: Exercise**

This shows that polynomial transformability implies Turing-reducibility. The converse is false however. To see why this is not surprising consider the problem 3-SAT and $(3\text{-SAT})^c$. Clearly 3-SAT $\propto_T$ $(3\text{-SAT})^c$ since for any instance $I$ we could write its encoding on a $(3-\text{SAT})^c$-oracle tape and wait for the answer. Then we simply output yes if and only if the oracle responds with the empty string $\epsilon$ (NO). On the other hand, the task of finding an actual transformation of each instance $3-$SAT to an instance of $(3-\text{SAT})^c$ seems non-trivial (and in fact none is known).

We can now extend the notion of $\mathcal{NP}$-completeness to capture the notion of a search problem being *at least* as hard as any problem in $\mathcal{NP}$.

**Definition 13.2** *A search problem* $\Pi$ *is* <u>*NP-Hard*</u> *if there exists some* $\mathcal{NP}$*-complete problem* $\Pi'$ *such that* $\Pi' \propto_T \Pi$*.*

It is easily seen that if a problem is $\mathcal{NP}$-Hard, then it can be solved by a polytime algorithm only if $\mathcal{P} = \mathcal{NP}$.

We could continue the above approach to develop a generalised complexity theory relative to a fixed language/string relation/search problem $R$. For example, we could define decision problem classes $\mathcal{P}^R, \mathcal{NP}^R$ based on whether there exist polytime $R$-oracle TM (nondeterministic) algorithms. We could then define polynomial transformability exactly as before to obtain the notion of $\mathcal{NP}^R$-completeness. The following have been shown that with these definitions:

- There is a decision problem $\Pi$ such that $\mathcal{P}^{R_\Pi} = \mathcal{NP}^{R_\Pi}$
- There is a decision problem $\Gamma$ such that $\mathcal{P}^{R_\Gamma} \neq \mathcal{NP}^{R_\Gamma}$.

Another direction is to call a decision problem *reducible NP-complete* if (a) $\Pi \in \mathcal{NP}$ and (b) $\Pi' \propto_T \Pi$ for each $\Pi' \in \mathcal{NP}$. Theorem 8 evidently implies that any $\mathcal{NP}$-complete problem is also reducible $\mathcal{NP}$-complete . The converse of this remains an intriguing open question.

## 14 Binary Search and an Application to Turing-Reducibility

In this section we speak often of a finite set $A$ with positive integer weights $s : A \to \mathbf{Z}^+$ associated with its members. If $A' \subseteq A$ we write $s(A')$ to denote the quantity $\sum_{a \in A'} s(a)$.

Consider the number problem:

**Problem:** PARTITION

**Instance:** A finite set $A$, and weights $s : A \to \mathbf{Z}^+$.

**Question:** Does there exist a subset $A'$ such that $s(A') = s(A - A')$?

PARTITION is $\mathcal{NP}$-complete but there is a pseudo-polytime algorithm which solves it. We now consider a related number problem:

**Problem:** $k^{th}$ Largest Subset (KLS)

**Instance:** A finite set $A$, weights $s : A \to \mathbf{Z}^+$ and positive integers $B, k$ with $B \leq s(A)$, $k \leq 2^{|A|}$.

**Question:** Do there exist **at least** $k$ subsets $A'$ such that $s(A') \leq B$.

This problem is a relative to PARTITION and would thus appear not to be in $\mathcal{P}$ , but it is also not clear whether it is even in $\mathcal{NP}$ ! The normal way of showing this would involve guessing $k$ subsets and checking that they verify the YES answer. The problem is that a certificate containing $k$ subsets would not have size bounded (in general) by a polynomial in $|I| = O(|A|\lceil log(s(A))\rceil + \lceil logB\rceil + \lceil logk\rceil)$.

In addition, there is no known transformation of an $\mathcal{NP}$-complete problem to KLS. There is, however, an easily described Turing-reduction of PARTITION to KLS which illustrates the power of using Turing-reducibilty. The reduction uses *binary search*, a very general method often applicable to optimisation problems. We imagine that $kls(A, s, B, k)$ is an oracle (or *subroutine*) which solves KLS in one step (or polynomially bounded steps). Here then, is a polytime KLS-oracle algorithm to solve PARTITION. (Before starting, note that a yes answer to the partition problem is equivalent to the existence of a set $A'$ such that $s(A') = \frac{s(A)}{2}$.)

Step 1. Compute $s(A)$.
If $s(A)$ is odd, then enter $q_N$.
Otherwise, set $b := \frac{s(A)}{2}$.

**Binary Search:** Find the number $M$ of sets $A'$ with $s(A') \leq b$.

Step 2. $MIN := 0$; $MAX := 2^{|A|}$.

Step 3. If $MAX - MIN = 1$, then $M := MIN$ and halt binary search (go to Step 5)

Step 4. $TEMP := \frac{(MAX+MIN)}{2}$
Call $kls(A, s, b, TEMP)$
If Answer is YES, $MIN := TEMP$ and Goto Step 3.
Otherwise, $MAX := TEMP$ and Goto Step 3.

**End Binary Search**

Step 5. Call $kls(A, s, b-1, M)$

Step 6. If YES, then enter $q_N$, otherwise enter $q_Y$.

It is easy to check that the above algorithm has polynomial running time as long as the subroutine $kls()$ can be solved by a polytime algorithm (or 1-step oracle!).

**Exercise:** Show that at each step of the algorithm, the number $M$ of sets $A'$ with $s(A') \leq b$ satisfies $MIN \leq M < MAX$.

The above result implies that at Step 5, we have determined through binary search that there are precisely $M$ sets $A'$ such that $s(A') \leq \frac{s(A)}{2} = b$. Thus Step 6 checks to see if all of those sets also satisfy $s(A') \leq b-1$. If they do, then there is no solution to PARTITION.

**Corollary 9** KLS *is* $\mathcal{NP}$ *-Hard.*

## 15 Strong NP-completeness and Pseudo Polynomiality

We informally refer to a *number problem* as one whose instances contain occurrences of (nonnegative usually) integers. For an instance $I$ we denote by $max(I)$ the size of a largest integer occurring in $I$. A *pseudo polynomial time* algorithm is one whose running time is bounded

by some polynomial in the quantities $|I|$ and $max(I)$ (as opposed to just $|I|$), such as $50|I|^2(max(I))^5$.

An example of a problem with no known polytime algorithm but for which there is an easy pseudo polytime algorithm is:

**Problem:** Compositie Number (COMP)

**Instance:** Positive integer $N$.

**Question:** Is $N$ composite? (i.e., not prime).

An instance $I$ for the problem is an integer $N$ and so the size of $I$ is the length of an encoding of $N$ which is $O(\log N)$. Thus a polytime algorithm for the problem must terminate after $O((\log N)^k)$ steps for some constant $k$. *No such algorithm is known for* COMP.

On the other hand, consider the following trivial algorithm:

Step 1. For $i = 2$ to $N - 1$.

Step 1.1 Check if $N$ is divisible by $i$
(N.B. this **can** be done in $O(logN)$ time).

Step 1.2 If YES then enter $q_Y$.

End For loop

Step 2. Enter $q_N$.

The algorithm clearly solves COMP since if the algorithm does not enter $q_Y$ in one of the iterations of Step 1, then $N$ is not divisible by any $i < N$ other than 1 and hence is prime. Moreover, the number of steps taken by the algorithm is
(no. times step 1 is performed) $\times$ (1+(no. steps taken in Step 1.1)) + (no. times Step 2 is performed).

Since Step 1 is performed at most $N-1$ times and Step 1.1 can be done in $O(logN)$ time, the algorithm takes at most $O(NlogN) = O(max(I)|I|)$ steps and hence is pseudo polynomial time.

A number problem $\Pi$ is *strongly NP-complete* if there exists a polynomial $p$ such that the following problem is $\mathcal{NP}$-complete .

**Problem:** $p$-restricted $\Pi$

**Instance:** An instance $I$ of $\Pi$ for which $max(I) \leq p(|I|)$.

**Question:** Same as $\Pi$.

Examples of strongly NP-complete problems include TSP, MAX CUT and BIN PACKING.

**Exercise 15.1** *Show that* TSP *is strongly* $\mathcal{NP}$*-complete .*

## 16 COMP $\in \mathcal{NP} \cap co\mathcal{NP}$

We have already seen that COMP $\in \mathcal{NP}$ since for given YES-instance $N$ of COMP, we need only exhibit two proper facts $n_1, n_2$ as a certificate and then there is a polytime algorithm which does arithemtic to verify that $N = n_1 n_2$. In order to show that COMP $\in co\mathcal{NP}$ we will have to delve a little into some elementary number theory.

We start by noting that any positive integer $N$ can be expressed as

$$N = \Pi_{i=1}^{n} p_i^{a_i} \tag{4}$$

where the $p_i$'s are distinct prime numbers. This is called $N$'s *canonical factorisation*. We denote by $gcd(n, m)$ the greatest common divisor of two positive integers $n, m$ and the integers are called *relatively prime* if $gcd(n, m) = 1$. Denote by $R_N$, the set of integers $m \leq N$ which are relatively prime to $N$. The *Euler* $\Phi$-function is defined as $\Phi(N) = |R_N|$. For example $\Phi(10) = 4$ since $R_{10} = \{1, 3, 7, 9\}$. There is a closed expression for $\Phi$ based on canonical factorisations.

**Exercise 16.1** *For $N$ with canonical factorisation as in (4) show that*

$$\Phi(N) = N\Pi_{i=1}^{n}(1 - \frac{1}{p_i}).$$

It is another relatively simple exercise to show:

$$\Phi(N) \leq (N-1) \text{ and } \Phi(N) = N - 1 \text{ if and only if } N \text{ is prime.} \tag{5}$$

Consider the problem:

**Instance:** Two positive integers $N, M$.
**Find:** Integers $s, t$ such that $gcd(N, M) = sN + tM$. (6)

It is not obvious that such integers $s, t$ need exist but it is in fact a consequence of the *Euclid Algorithm* that they do. We refer the reader to the exercises for a description of the algorithm but for now it suffices to

accept that there is a polytime algorithm which solves the above search problem.

For a positive integer $z$ we denote by $[z]_N$ the integer remainder upon dividing $N$ by $z$. For example $[z]_2$ is 1 if and only if $z$ is odd and otherwise it is 0. One consequence of Euclid's Algorithm for (6) is that $(R_N, \times_N)$ forms a multiplicative group, where $\times_N$ denotes $modN$ multiplication, i.e., $x \times_N y = [xy]_N$. It is clear that $R_N$ has an identity, namely 1 itself, since $1 \times_N x = x$ for each $x \in R_N$. It remains to show that each $m \in R_N$ has an inverse. But this is true since $gcd(m, N) = 1$ and so by Euclid's Algorithm there exists $s, t$ such that $1 = sN + tm$. Thus $tm = 1 - sN \equiv 1(modN)$. Thus $[t]_N = m^{-1}$ as desired.

We need only two simple facts about groups. Recall that the *order* of a group $G$, denoted by $|G|$, is the number of elements in it and that for each $x \in G$, the *order* of $x$ is the smallest integer $p$ such that $x^p$ is the identity. We need the following basic facts:

**Lemma 10** *If $G$ is a finite group, then each $x \in G$ has finite order, and its order, $p$ say, divides $|G|$. Moreover, $x, x^2, \ldots, x^p$ are distinct elements of $G$.*

An immediate consequence of this which we will need later.

**Theorem 11 (Fermat's Little Theorem)** *If $N$ is prime and $x \in R_N$, then $x^{N-1} \equiv 1(modN)$.*

**Proof:** Suppose that $x$ has order $n$. Then by Lemma 10, $N - 1 = bn$ for some integer $b$ and so $x^{N-1} = (x^n)^b \equiv 1^b(modN)$ as required. □

In fact there is a kind of converse to this.

**Fact 12** *If $x^d \equiv 1(modN)$ for some $d \geq 1$, where $x \in \{1, \ldots, N-1\}$, then $x \in R_N$.*

**Proof:** Suppose that $N = ab, x = ac$. If $x^d \equiv 1(modN)$, then $x^d = 1 + tN$ for some integer $t$. Thus $1 = x^d - tN$ which in turn equals $(ac)^d - tab$. This last term is a multiple of $a$ and so can equal 1 only if $a = 1$. Thus 1 can be the only common divisor of $N$ and $x$. □

We are now ready to give the one result needed to show COMP ∈coNP .

**Lemma 13** *Let $N > 2$ be an integer and suppose there exists a positive integer $x$ such that $x^{N-1} \equiv 1(modN)$. If for all divisors $d$ of $N-1$, $1 < d < N-1$, $x^d \not\equiv 1(modN)$, then $N$ is prime.*

**Proof:** Let $x$ have order $n$. By Lemma 5, $x, x^2, x^3, \ldots x^n$ are distinct and $x^n \equiv 1(modN)$. Thus $x^{n+i} \equiv x^i(modN)$ for each $i$. It follows that $x^i \equiv 1(modN)$ if and only if $i = kn$ for some integer. But by hypothesis, $x^(N-1) \equiv 1(modN)$ and so $N - 1 = kn$ for some integer $k$. That is the order $n$, of $x$, divides $N-1$. By assumption, $x^d \not\equiv 1(modN)$ for any proper divisor $d$ of $N-1$, and so $x$ must have order exactly $N-1$. Thus by the second part of Lemma 10, $x, x^2, \ldots, x^{N-1}$ are distinct elements and for each $i$, $(x^i)^{N-1} \equiv (x^{N-1})^i \equiv 1(modN)$ and so each of these elements is in $R_N$ by Fact 12. Thus $\Phi(N) = |R_N| = (N-1)$ and so $N$ is prime by (5). □

We can now prove the main result.

**Theorem 14** COMP $\in co\mathcal{NP}$ .

**Proof:** We now describe a succinct certificate $\mathcal{C}(N)$ for '$N$ is prime', i.e., a NO-instance of COMP. It is a little different from previous certificates we have seen since it is *recursive*. That is to say, it contains certificates for other NO-instances of COMP. We will assume that $N$ is odd since if $N = 2$ we check immediately that it is prime, and otherwise $N$ is clearly not prime. The certificate $\mathcal{C}(N)$ contains:

- a canonical factorisation of $N-1$, i.e., the primes $p_i$ and exponents $a_i$,
- the certificates $\mathcal{C}(p_i)$ for each of these smaller primes,
- a positive integer $x$.

We leave as an exercise that there is a polynomial bound on the length of $\mathcal{C}(N)$. To check that $N$ is prime, an algorithm now need only verify that (i) the canonical factorisation is correct, (ii) $x^{N-1} \equiv 1(modN)$ and (iii) no proper divisor $d$ of $N-1$ satisfies $x^d \equiv 1(modN)$. This last step, however, is potentially problematic, i.e., could lead to a nonpolynomially

bounded number of steps. It turns out that we need not check every divisor $d$ since if $N-1$ has a divisor $d$ such that $x^d \equiv 1(modN)$, then it has one of the form $d = (N-1)/p_i$, for some $i$ (see Exercise 16.2). The checking algorithm now performs the following steps:

1. Check that $N - 1 = \Pi_{i=1}^{n} p_i^{a_i}$.
2. Recursively check each $\mathcal{C}(p_i)$ to verify that $p_i$ is a prime.
3. For each $i = 1, \ldots, n$ check that $x^{\frac{N-1}{p_i}} \not\equiv 1(modN)$.
4. Check that $x^{N-1} \equiv 1(modN)$.

We analyse the number of steps performed by the above checking algorithm where a step will be measured as the number of times we perform an operation of the form 'Is $x^d \equiv 1(modN)$' or 'Is $N - 1 = \Pi p_i^{a_i}$'. (We leave as an exercise that each of these types of steps is also computable in time $O(logN)$.) Thus the number of steps performed in each of 1. and 4. is 1 and in 3. is $n$, where $n$ is the number of prime divisors of $N-1$. Let $c(N)$ be the number of steps dones by the whole algorithm, then since it is recursive (it calls itself in 2.) we have

$$c(N) = n + \sum_{i=1}^{n} c(p_i) + 2.$$

We now show that $c(N) = O(logN)$. Note that we can rewrite this as

$$g(N) = \sum_{i=2}^{n} g(p_i) + 4$$

where $g(N) = c(N) + 1$ and where we assume that $p_1 = 2$ and is verifiable as prime in one step (recall that $N-1$ is even). We show that $g(N) \leq 4log(N)$ for all $N$. This is clear for $N = 2$ so assume that this holds for all primes less than $N$.

$$\begin{aligned} c(N) \leq & \ (\Sigma_{i=2}^{n} 4log(p_i)) + 4 \\ = & \ 4(log(\Pi_{i=2}^{n} p_i) + 1) \\ \leq & \ 4(log(N/2) + 1) \leq 4log(N), \end{aligned}$$

as desired. Since $log(N)$ is the size of an instance $N$ we have thus verified that the checker algorithm performs $O(log(N)$ operations of the type described above. Since each of these operations is linear in the size of the input, the running time of the checker algorithm is $O(|I|^2)$. $\Box$

**Exercise 16.2** *For positive integers $x, N$, show that if $N-1$ has a divisor $d$ such that $x^d \equiv 1(modN)$, then $N-1$ has such a divisor of the form $(N-1)/p_i$ for some $i = 1, \ldots, n$ where $p_i$'s are given by the canonical factorisation of $N-1$.*

**Exercise 16.3** *Write the certificate $\mathcal{C}(17)$.*

**Exercise 16.4** *Find all integers $x \in R_{19}$ which satisfy the hypotheses of Theorem 13.*

**Exercise 16.5** *Give a polynomial $q$ and for which the number of bits needed to encode $\mathcal{C}(N)$ is at most $q(log(N))$.*

**Exercise 16.6 Euclid Algorithm**: *The following algorithm solves the problem (6).*

EUCLID*(N,M)*

*If $M = 0$, then $t = 0, s = 1$ and $gcd(N, M) = N$*

*Else call* EUCLID*(M,N mod M)*

*Let $s', t'$ be the output from this. Then (1) $t = s' - \lfloor N/M \rfloor t'$ (2) $s = t'$*
*Output (s,t)*

*Show that it does in fact solve (6). Prove that* EUCLID *is a polytime algorithm.*

# 17 Alternatives for $\mathcal{NP}$ -Hard Problems

What if we are faced with an $\mathcal{NP}$ -Hard problem which we simply must solve? Since it is unlikely that the problem may be solved be a fast algorithm, our alternative seems only to either relax what we mean by algorithm (e.g., allow some human interaction) or to relax what we mean by 'solve' or 'fast'. Approaches of the latter variety fall into three main categories:

- Algorithms which give approximate answers within a certain error guarantee.
- Algorithms which almost always give the correct answer.
- Algorithms which almost always run fast.

We describe the first two of these in more detail in the following sections.

## 17.1 Approximation Algorithms

We consider the bin packing optimisation problem.

**Problem:** BIN-OPT
**Instance:** Set $A$, sizes $s : A \to \mathbf{Z}^+$ and capacity $U$.
**Find:** A minimum number $k$ such that $A$ can be partitioned into sets $B_1, B_2, \ldots, B_k$ such that $\sum_{a \in B_i} s(a) \leq U$ for each $i$.

Suppose that the elements of $A$ have been put in some order $a_1, a_2, \ldots, a_n$ and imagine that there exists an infinite queue of bins $C_1, C_2, \ldots, C_i, \ldots$ in which to place the $a_i$'s. We consider the so-called *First Fit* (FF) algorithm which gives some bin packing (not necessarily optimal) for the set $A$. The algorithm starts by placing $a_1$ into container $C_1$ (it is assumed that none of the $a_i$'s has size greater than the bin capacity $U$). It then examines whether $a_2$ will fit into $C_1$ together with $a_1$, that is, it checks if $s(a_2) > U - s(a_1)$. If there is still room, then $a_2$ goes into $C_1$,

otherwise a new bin is started and $a_2$ is placed into $C_2$. We continue in a similar fashion finding containers for $a_3, a_4, \ldots, a_n$. At each step, if we are looking for a home for $a_i$, it is placed in the first container which still has space for it. Thus we check $C_1, C_2, \ldots$ etc. until we find a container $C_j$ with at least $s(a_i)$ empty space remaining and then $a_i$ is placed in $C_j$.

For an instance $I$, let $FF(I)$ denote the number of non-empty containers which are created by $FF$ [4] and let $OPT(I)$ denote the optimal solution to BIN-OPT. Our goal is to show that $FF$ satisfies some guaranteed performance level. To see this, note that after $FF$ has finished, no two containers $C_i, C_j$ could be at most half full. For otherwise if $i < j$, we could place the contents of $C_j$ into $C_i$ and so the algorithm could not have placed the contents of $C_j$ by the 'first fit' rule. Thus we must have that the contents of each of $C_1, C_2, \ldots, C_{FF(I)-1}$ is greater than $\frac{U}{2}$ and so

$$\sum_{a \in A} s(a) \geq \sum_{i=1}^{FF(I)-1} \text{contents of } C_i > (FF(I) - 1)\frac{U}{2}.$$

Thus $FF(I) - 1 < \frac{2s(A)}{U}$ and so

$$FF(I) < \lceil (2s(A))/U \rceil. \tag{7}$$

On the other hand, even if an optimal solution had each bin filled perfectly to the top, then we would need $\lceil \frac{s(A)}{U} \rceil$ bins. This together with (7) shows that $FF(I) < 2OPT(I)$, or $\frac{FF(I)}{OPT(I)} < 2$. Thus we can give a guarantee as to how bad $FF$'s solution can be relative to the optimum. **Exercise:** Show that for each instance $I$ of BIN-OPT, there is an ordering $a_1, a_2, \ldots, a_n$ such that when FF is applied to this ordering we get $FF(I) = OPT(I)$.

A *minimisation problem* $\Pi$ is similar to a search problem except that for each instance $I$ there will be a *feasible set* $F_\Pi(I)$ and a a polynomially

[4] Different orderings of the $a_i$'s may lead to different solutions of course, but for our purposes we may assume that the input has even been specified with an order on the members of $A$. Note that we only wish to analyse the worst case behaviour of FF.

computable objective function $c$ which assigns an integer to each feasible solution in any given instance. These in turn define the *optimal set* $O_\Pi(I)$ [5] which consists of $\{s \in F_\Pi(I) : c(s) \text{ is minimised.}\}$. We define similarly a maximisation problem and an *optimisation problem* indicates a problem of either of these two types. An *approximation algorithm* for an optimisation problem is one which given an instance of the problem, halts with a feasible solution. The *absolute performance ratio* on instance $I$, denoted by $R_A(I)$, of an approximation algorithm $A$ for a minimisation problem is defined as $\frac{A(I)}{OPT(I)}$. For a maximisation problem, the absolute performance ratio is defined as the inverse $\frac{OPT(I)}{A(I)}$. Note that for any optimisation problem, this ratio has value at least 1. We have shown that $R_{FF}(I) < 2$ for every instance $I$.

The *relative error* of an approximation algorithm (for a minimisation problem) which is

$$(A(I) - OPT(I))/OPT(I).$$

An An $\epsilon$-*approximate* algorithm is one for which the relative error is at most $\epsilon$ for all instances $I$. Note that we have shown that $FF$ is a 1-approximate algorithm for BIN-OPT.

So far we have only spoken about performance 'ratios'. What about constant difference error bounds? In fact some elegant negative results can be proven in this regard. We consider the knapsack problem:

**Problem:** Knapsack Optimisation (KS-OPT)

**Instance:** A set $A$, sizes $s : A \to \mathbf{Z}^+$, capacity $U$, and values $v : A \to \mathbf{Z}^+$.

**Find:** A subset $A' \subset A$ which has maximum value $v(A')$ subject to satisfying $s(A') \leq U$.

KS-OPT is NP-Hard. Moreover, we can show that it has no polytime approximation algorithm with constant error guarantee, **unless** $\mathcal{P} = \mathcal{NP}$ .

[5] This set often corresponds to the feasible set of a search problem

**Theorem 15** *If $\mathcal{P} \neq \mathcal{NP}$ , then there is no polytime approximation algorithm $A$ for* KS-OPT *and integer $k$ such that:*

$$|A(I) - OPT(I)| \leq k$$

*for all instances $I$.*

**Exercise 17.1** *Prove it.*

**Hint:** Suppose $A$ is such an algorithm for some integer $k$. Show that $A$ must in fact solve (not just approximate) KS-OPT by replacing each instance $I$ by a new one $I'$ where the values have been scaled up (e.g., $v'(a) = (k+1)v(a)$ for each $a \in A$).

## 18 Randomised Algorithms

As we have mentioned, there is no known polytime algorithm which solves COMP, but here is a possible next best thing. Suppose that $A$ is a polytime algorithm which takes positive integers $b < N$ as input such that $A$ satisfies:

1. $A$ always terminates with YES or NO (= maybe).
2. If output is YES, then $N$ is composite.
3. If $N$ is composite, then the output is YES for at least one half of the $b$'s in $\{1, 2, 3, \ldots, N-1\}$.

Thus if $A$ outputs YES, then we know that the answer is correct but if it outputs NO, the answer may be right or wrong. We show later that such an algorithm exists but for now suppose that we are given $A$ and that our machine also has access to a random number generator. Consider the following procedure:

For $i = 1$ to 100
    Pick an integer $b$ at random from $\{1, \ldots, N-1\}$.
    Call $A$ with input $N, b$.
    If $A$ outputs YES, then enter $q_Y$ and quit.
End For Loop.
Enter state $q_N$.

We know that the description of $A$ guarantees that if we end in state $q_Y$, then the answer is correct. Moreover, if $N$ is prime, then the answer is correct. Thus the algorithm only outputs a wrong answer if $N$ is composite but in each of the 100 calls to the algorithm $A$, the output was NO. Since in each iteration of the for loop, $b$ is picked at random, the probability that $A$ answers NO when $N$ is actually composite is **at most** $\frac{1}{2}$. This is by condition 3. for $A$. Thus the probability that we randomly pick 100 $b's$ which give the wrong answer is at most $(\frac{1}{2})^{100}$. This is getting pretty small and in fact for any $\epsilon > 0$ we can construct an algorithm $R_\epsilon$ for COMP which is wrong with probability at most $\epsilon$.

This is done simply by choosing $k$ large enough so that $(\frac{1}{2})^k \leq \epsilon$ and then repeating the above procedure $k$ times instead of 100.

A *random TM* is basically the same as a TM except that it has access to a *random bit stream* (or has a man which flips coins inside it if you prefer). We allow the machine to enter a *stream-read* state in which it reads bits from the bit stream and the bits $0 - 1$ each occur with probability $\frac{1}{2}$. Each read requires one basic operation, or step, for the random TM.

A *randomised algorithm* $A$ (i.e., random TM program) *solves* a decision problem $D(L)$ if

1. $A$ always halts
2. If $A$ halts in $q_Y$, then $x \in L$.
3. If $x \in L$, then $A$ halts in $q_Y$ with probability at least $\frac{1}{2}$. [6]

We denote by $\mathcal{RP}$ the class of decision problems which are solved by a polytime randomised algorithm. We then have the following result whose proof we leave as a simple exercise.

**Theorem 16** $\mathcal{P} \subseteq \mathcal{RP} \subseteq \mathcal{NP}$ .

It is strongly believed that $\mathcal{RP} \neq \mathcal{NP}$ and at least some people suspect that $\mathcal{P} \neq \mathcal{RP}$ . In fact the archetype of a problem in $\mathcal{RP}$ (see next section) but not known to be in $\mathcal{P}$ is COMP. The following result is analogous to Theorem 5.

**Theorem 17** *If any $\mathcal{NP}$-complete problem is in $\mathcal{RP}$ , then $\mathcal{RP} = \mathcal{NP}$ .*

**Exercise 18.1** *Prove Theorem 17.*

**Exercise 18.2** *If I believe that $\mathcal{P} \neq \mathcal{RP}$ , what can you say about my belief in the veracity of Conjecture 6.1. Explain.*

[6] There is nothing magical about $\frac{1}{2}$; any positive fraction would do.

### 18.1 The Solovay-Strassen Randomised Algorithm for COMP

We start immediately with the description of the Solovay-Strassen algorithm which given positive integers $b < N$ does the following:

**Step 1.** Find the *odd part* $M$ of $N-1$, i.e., $N-1 = 2^q M$, where $M$ is odd.

**Step 2.** If either (a) $b^M \equiv 1(modN)$ or
(b) $b^{M2^i} \equiv -1(modN)$ for some $i = 0, 1, \ldots, q$,
then enter $q_N$, else enter $q_Y$.

To prove that if $N$ is composite, then there are at least $\frac{1}{2}$ the $b$'s which cause the algorithm to output YES requires a little more number theory (although not too much), and so we omit this. We will be satisfied in showing only that if the algorithm halts in $q_Y$, then $N$ is composite. We will use Fermat's Little Theorem and one last fact.

**Lemma 18** *If $N$ is prime, then $x^2 \equiv 1(modN)$ can be true only if $x \equiv \pm 1(modN)$.*

**Proof:** If $x^2 - 1 \equiv 0(modN)$, then $N|(x+1)(x-1)$ and so since $N$ is prime, $N|(x+1)$ or $N|(x-1)$ (seen by considering the canonical factorisation of $(x+1)(x-1)$). Thus $x \equiv 1$ or $-1(modN)$, as desired. □

Now suppose that the S-S algorithm outputs YES but the input $N$ was prime. We claim that $b^{M2^i} \equiv 1(modN)$ for each $i = 0, 1, 2, \ldots q$. This is proved by reverse induction on $i$. For $i = q$, this follows from Fermat's Little Theorem since $M2^i = N - 1$. Now we show if it is true for $i$, then it is also true for $i-1$. This is because $b^{M2^i} = (b^{M2^{i-1}})^2$ and so $x = b^{M2^{i-1}}$ is a solution to $x^2 \equiv 1(modN)$ since our claim holds for $i$. Now by Lemma 18 this can be true only if $x \equiv \pm 1(modN)$. Moreover, $x \not\equiv -1(modN)$ for then S-S would have outputted NO. Thus $x \equiv 1(modN)$, and the claim follows by induction. In particular, taking $i = 0$ this shows that $b^M \equiv 1(modN)$ and so the algorithm should have again outputted NO, a contradiction. Thus the S-S algorithm is indeed correct if it outputs YES.

Note that the algorithm merely asserts whether $N$ is composite or not, it does not find any factors! This latter search problem may indeed be more difficult.

## 19 Circuit Complexity

Denote by $B_n$ the set of boolean functions $f : \{0,1\}^n \to \{0,1\}$. Let $\Omega$ be any finite set of boolean functions and for each $\omega \in \Omega$ denote by $n(\omega)$ the integer $n$ such that $\omega \in B_n$.

An $\Omega$-*circuit* $C$ on (fixed) input boolean variables $x_1, \ldots, x_n$ consists of a sequence $(G_1, G_2, \ldots, G_k)$ of *gates*. A gate is characterised by:

- its *type* $\omega_i \in \Omega$ and
- its *input* which is a sequence of *predecessors* $(P_1, P_2, \ldots, P_{n_i})$, where $n_i \equiv n(\omega_i)$ and each $P_i \in \{1, 0, x_1, x_2, \ldots, x_n, G_1, \ldots, G_{i-1}\}$.

For a given input $\overline{r} = (r_1, \ldots, r_n)$ where each $r_i \in \{0,1\}$, i.e., assignment of values to the input variables $(x_1 = r_1, \ldots, x_n = r_n)$, we define the *result* of $x_i$, denoted by $res_{x_i}(\overline{r})$ to be the value $r_i$. For each gate $G_i$, we recursively define $res_{G_i}(\overline{r}) = \omega_i(res_{P_1}(\overline{r}), \ldots, res_{P_{n_i}}(\overline{r}))$ where obviously $res_y(\overline{r}) = y$ if $y \in \{0,1\}$. The result of the circuit $res_C(\overline{r})$ is $res_{G_k}(\overline{r})$. The function $f \in B_n$ *computed* by $C$ is $f(\overline{r}) \equiv res_C(\overline{r})$.

The *size* of a circuit is its number of gates $k$. The *complexity* of a function $f$ *over* $\Omega$, denoted by $\kappa_\Omega(f)$ is the smallest size of an $\Omega$-circuit computing $f$. [7]

A *complete basis* is a set of functions $\Omega$ such that every boolean function is computed by some $\Omega$-circuit.

**Fact 19** *$\Omega = \{\vee, \wedge, \neg\}$ is a complete basis.*

**Exercise 19.1** *Show that the function $g(x_1, x_2) = 0$ if $x_1 = x_2 = 1$ and 0 otherwise, is not computed by any $\{\vee, \wedge\}$-circuit. And hence $\{\vee, \wedge\}$ is not a complete basis.*

[7] The *depth* of a circuit is the length of a longest input-path, i.e., $G_{a_1}, G_{a_2}, \ldots, G_{a_p}$ such that $G_{a_i}$ is a predecessor of $G_{a_{i+1}}$ for each $i < p$. There has also been considerable work done in the analysis of maximum depth of circuits.

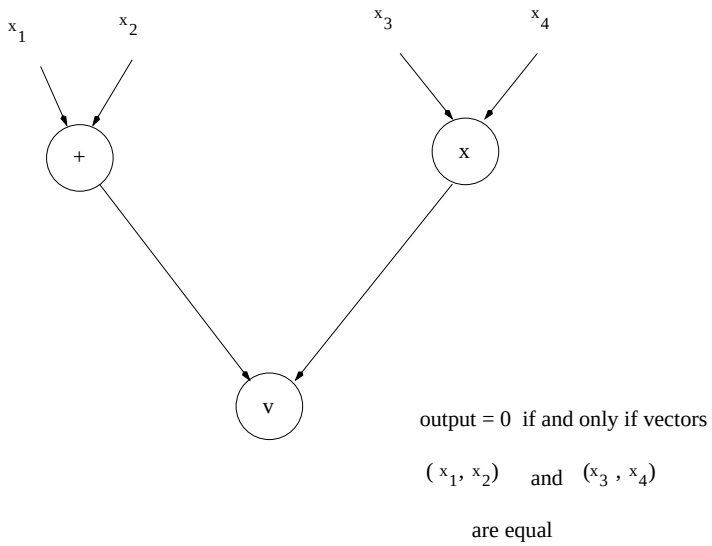


Figure 10: $\Omega = \{\vee, \oplus\}$.

We will usually assume that $\Omega = \{\vee, \wedge, \neg\}$ and will henceforth dispense with explicit reference to the underlying set of functions.

Note that we can associate any sequence $\mathcal{F} = (f_1, f_2, \ldots)$ of functions $f_i \in B_i$ with a language $L_{\mathcal{F}} \in \{0,1\}^*$ whose strings of length $n$ are precisely those $x_1 x_2 \ldots x_n$ for which $f_n(x_1, x_2, \ldots, x_n) = 1$. This association is invertible and we denote by $\mathcal{F}_L$ the sequence of functions associated with a language $L$.

Call a sequence $\mathcal{F}$ *compact* if it is computed by a polynomially bounded sequence of circuits, i.e., there exists a polynomial $p$ such that $\kappa(f_n) \leq p(n)$ for all $n$. It is **not** always true that if $\mathcal{F}$ is compact, then the decision problem for $L_{\mathcal{F}}$ is in $\mathcal{P}$ . However, we do have the following result.

**Theorem 20** *If decision problem $D(L)$ is solved by an algorithm with running time $T(n)$, then the complexity of the $n^{th}$ function in $\mathcal{F}_L$ is $O(T^2(n))$.*

The proof of this describes how to 'simulate' a TM program by using circuits, i.e., for each $n$ we find a circuit on $n$ variables whose result is 1 if and only if the TM enters $q_Y$ on a corresponding input of length $n$.

**Corollary 21** *If $D(L) \in \mathcal{P}$ , then $\mathcal{F}_L$ is compact.*

This result is potentially useful since it implies that lower bounds for the complexity of circuits also give rise to lower bounds on algorithm running times. We now describe one such lower bound.

We say that *almost every* (a.e.) function has property $P$ if $lim_{n\to\infty}|\{f \in B_n : f \text{ has property P}\}|/|B_n| = 1$.

**Theorem 22** *Almost every boolean function has complexity at least $2^n/n$.*

**Proof:** □

Perhaps on a more positive note, we have the following upper bound.

**Theorem 23** *There exists a function $g(n)$ such that*

1. *$\kappa(f) \leq (1 + g(n))2^n/n$ for all $f \in B_n$, and*
2. *$\lim_{n\to\infty} g(n) = 0$.*

**Corollary 24** *There exists $N$ such that for each $n \geq N$, $\kappa(f) \leq 2^{n+1}/n$ if $f \in B_n$.*

**Exercises**

**Exercise 19.2** *Prove Corollary 21 using Theorem 20.*

**Exercise 19.3** *Prove Corollary 24 using Theorem 23 and then show that there is a constant $C$ such that $\kappa(f) \leq C2^n/n$ for all $f \in B_n$.*

**Exercise 19.4** *A. Use Theorems 20 and 22 to show that there exists a language $L_0$ whose decision problem is not solved by any polytime algorithm.*
*B. Why does this not necessarily imply that $\mathcal{P} \neq \mathcal{NP}$ ?*
*C. If $\mathcal{P} \neq \mathcal{NP}$ , then what can you say about any nondeterministic algorithm which solves $D(L_0)$?*

**Exercise 19.5 (For those with sufficient background in analysis)** *Show that almost all languages are not in $\mathcal{NP}$ . This requires an understanding of the uniform measure on the space $\Pi_{n=1}^{\infty}\Sigma^n$.*

**Exercise 19.6** *Prove Fact 19*

**Exercise 19.7** *(Harder) Prove Theorem 22*

**Exercise 19.8** *Design a circuit on variables $x_1, x_2, \ldots, x_n, y_1, \ldots, y_n$ which outputs 1 if and only if the vectors satisfy $(x_1, \ldots, x_n) \leq_L (y_1, \ldots, y_n)$ where $\leq_L$ denotes lexicographic order. What is the complexity of your circuit as a function of $n$?*

**Exercise 19.9** *Design a circuit which given inputs $x_1, \ldots, x_n$, outputs the binary representation of $\sum x_i$. Note that this circuit is slightly different since it does not compute a single output.*

**Exercise 19.10** *A language $L$ is <u>capped</u> if for each $n$, there exists $f_n \in B_n$ such that an $n$-string $x_1x_2 \ldots x_n$ is in $L$ if and only if $(x_1, x_2, \ldots, x_n) \leq_L (fn(x_1), f_n(x_1 + x_2), \ldots, f_n(\sum_{i=1}^{n} x_i))$. Show that $\mathcal{F}_L$ is compact.*

## 20 Communication Complexity

The topic of communication complexity has a similar flavour to circuit complexity except that the aim now is to minimise the *length* of communication, i.e., the number of transmitted bits. More explicitly, there is a finite set of processors with certain communication links between them. Each processor is initially with some *given input* from its range of *feasible inputs*. They wish to collectively compute some *output value* (assume for now this is a single bit) which is a function of their inputs. Moreover, if any machine knows each of the others' given inputs, then it can calculate the output value in one step.

It is assumed that the processors have unlimited computing capacity and that their local computation is free. Instead, they are charged for each bit of data transmitted between them. One can think of the example of the space shuttle repairing the Hubble telescope. At NASA there exist banks of computing resources and even on the shuttle itself there can be powerful computers. However, the cost of transmissions back and forth (partially due to redundancy needed to reduce transmission errors) is a much more stringent constraint.

For the sake of simplicity we shall analyse the basic approaches in the case of two processors joined by a transmission line. In communication complexity, the concept analogous to an algorithm is that of a *protocol*. This is a list of rules which specifies at each stage (i) who is to send the next bit, depending on previously transmitted bits and (ii) what the transmission bit should be, depending on that processor's input and on the bits previously transmitted between the processors. The computation is completed when one processor 'knows' the output bit. The *complexity* of a protocol is the number of bits transmitted in the worst case (i.e., for worst pair of inputs). Note that there is always the possibility to simply have one of the processors describe their input to the other. This is called the *trivial protocol*.

More formally, lets suppose that processor $P_1$ has feasible inputs $x_1, \ldots, x_m$ and $P_2$ has $y_1, \ldots, y_n$. We let $C$ be the so-called *communication matrix*

whose entry $c_{ij}$ is the output value when $P_1$ is given input $x_i$ and $P_2$, $y_j$. Each of the processors has their own copy of this matrix and hence can compute the output value if given the other processors input. Note that the encoding of these inputs or what they represent is not an issue since each processor has unlimited power. The inputs need only be thought of as being determined by a row number for processor $P_1$ or a column number for $P_2$. Thus the trivial protocol involves $P_1$ sending the index of its input row and this requires at most $log(m)$ bits. (N.B. If $n < m$, then the reverse trivial protocol would be more advantageous.)

A protocol must first determine who sends the first bit and the value of this bit depends on that processor's input. Using the above model, if $P_1$ transmits first, then the protocol must effectively specify a partition of the rows of $C$ into two classes and the transmitted bit tells $P_2$ which of these two classes contains $P_1$'s input row. Thus further computation is restricted to the submatrix $C_1$ consisting of the rows in this class. At stage $i$, the protocol specifies a partition of a submatrix $C_i$'s rows or columns (depending on who the protocol specifies to do the transmitting) and this leads to the next submatrix $C_{i+1}$. The computation is complete at stage $k$ if $C_k$ is *homogeneous*, i.e., consists entirely of 1's or of 0's. At this point, both processors know the value of the output bit. The complexity of the protocol is the largest value $k$ for which the computation is completed in stage $k$. The communication complexity of $C$, denoted by $\kappa(C)$ is then the value $k-1$. [8]

We have thus argued that the complexity of $C$ is determined by the combinatorial question of finding the minimum number of rounds needed to partition $C$ into homogeneous matrices, where at each round, the current submatrix is either split horizontally or vertically. Note that at each round, the maximum rank of a submatrix is decreased by at most a factor of two and hence we have the following lower bound.

**Lemma 25** *If $C$ has rank $r$, then $\kappa(C) \geq log(r)$.*

[8] Note that at the previous stage, $C_{k-1}$ would have consisted of an 'almost' homogenous matrix, i.e., it could be partitioned into two homogenous matrices. Thus if no row of $C_{k-1}$ contained both a 0 and a 1, then $P_1$ already knew the output value at stage $k-1$ by simply looking at its input row in $C_{k-1}$ (i.e., the output value was no longer dependent on which of $C_{k-1}$'s columns corresponded to $P_2$'s input). On the other hand, $P_2$ was still in the dark until $P_1$ sends the final bit indicating its input row.

This now implies that there are some examples where we can do no better than the trivial protocol.

**Corollary 26** *If $C$ has full row rank, then the trivial protocol minimises the number of transmitted bits in computing the output bit.*

A *communication problem* is a class $\mathcal{C}$ of $0-1$ matrices. For simplicity, assume these matrices are square and recall that the complexity of an $n \times n$ matrix is at most $log(n)$. We say that $\mathcal{C} \in \mathcal{P}_{comm}$ if there is a constant $c > 0$ such that $\kappa(C) \leq (loglog(n))^c$ for each $n \times n$ matrix $C \in \mathcal{C}$.

We now consider the concept of non-deterministic communication complexity. This can be understood by the example of the space shuttle running a check to determine if a piece of software needed to polish the Hubble lens is still correct. There is a current correct version of the program at ground control and so the space shuttle must check bit for bit that it contains the same version. The output to this communication problem is 1 if and only if the two versions are not identical. Note that if this is the case, then there is a simple proof or certificate of this fact. For example if Merlin could again lend a hand (or E.T. as suggested by Lovász) he may simply say: look at page 3, line 6, character 27. At this point NASA sees that it has an '$F$' and the shuttle checks to find it has a '$G$', and so both know that there is a mistake in the shuttle's version and so the output is 1. The formal notion of a *non-deterministic protocol* consists of a set $P$ of *proofs* together with some finite set of rules to instruct (depending on the inputs) each of the processors which proofs are to be accepted. These must then satisfy:

*For any given pair of inputs, the output bit is* 1 *if and only if there exists a proof $p \in P$ which is accepted by both $P_1$ and $P_2$.*

The set $P$ will consist of $0-1$ strings and the *complexity* of the protocol is the maximum length of these strings. The non-deterministic complexity of a communication matrix $C$ denoted by $\kappa_1(C)$ is then the minimum complexity of a non-deterministic protocol for $C$. It can be shown that:

**Fact 27** *The non-deterministic communication complexity of a matrix $C$ is the minimum number $t$ such that the 1's in $C$ can be covered by at most $2^t$ all-1 submatrices.*

Of course we are going to now introduce the class $\mathcal{NP}_{comm}$ which consists of all communication problems $\mathcal{C}$ for which there exists a constant $c > 0$ such that $\kappa_1(C) \leq (loglog(n))^c$ for each $n \times n$ matrix $C \in \mathcal{C}$. The notions of $\kappa_0(C)$ and $co\mathcal{NP}_{comm}$ are also defined analogously but the theory of communication does not continue in this pat fashion.

**Theorem 28**

- $\mathcal{P}_{comm} \neq \mathcal{NP}_{comm}$,
- $\mathcal{P}_{comm} = \mathcal{NP}_{comm} \cap co\mathcal{NP}_{comm}$.

This is all that we say about this elegant theory but we refer the reader to the excellent survey [**?**].