Complexity Theory

Johan Håstad
Department of Numerical Analysis and Computing Science
Royal Institute of Technology
S-100 44 Stockholm
SWEDEN
johanh@nada.kth.se

May 13, 2009

Contents

1 Preface

The present set of notes have grown out of a set of courses I have given at the Royal Institute of Technology. The courses have been given at an introductory graduate level, but also interested undergraduates have followed the courses.

The main idea of the course has been to give the broad picture of modern complexity theory. To define the basic complexity classes, give some examples of each complexity class and to prove the most standard relations. The set of notes does not contain the amount of detail wanted from a textbook. I have taken the liberty of skipping many boring details and tried to emphasize the ideas involved in the proofs. Probably in many places more details would be helpful and I would be grateful for hints on where this is the case.

Most of the notes are at a fairly introductory level but some of the section contain more advanced material. This is in particular true for the section on pseudorandom number generators and the proof that $IP = PSPACE$. Anyone getting stuck in these parts of the notes should not be disappointed.

These notes have benefited from feedback from colleagues who have taught courses based on this material. In particular I am grateful to Jens Lagergren and Ingrid Lindström. The students who have taken the courses together with other people have also helped me correct many errors. Sincere thanks to Jerker Andersson, Per Andersson, Lars Arvestad, Jörgen Backelin, Christer Berg, Christer Carlsson, Jan Frelin, Mikael Goldmann, Pelle Grape, Joachim Hollman, Andreas Jakobik, Wojtek Janczewski, Kai-Mikael Jää-Aro, Viggo Kann, Mats Näslund, and Peter Rosengren.

Finally, let me just note that there are probably many errors and inaccuracies remaining and for those I must take full responsibility.

2 Recursive Functions

One central question in computer science is the basic question:

What functions are computable by a computer?

Oddly enough, this question preceded the invention of the modern computer and thus it was originally phrased: "What functions are mechanically computable?" The word "mechanically" should here be interpreted as "by hand without really thinking". Several independent attempts to answer this question were made in the mid-1930's. One possible reason that several researchers independently came to consider this question is its close connections to the proof of Gödel's incompleteness theorem (Theorem 2.32) which was published in 1931.

Before we try to formalize the concept of a computable function, let us be precise about what we mean by a function. We will be considering functions from natural numbers ($\mathbf{N}=\{0, 1, 2\ldots\}$) to natural numbers. This might seem restrictive, but in fact it is not since we can code almost any type of object as a natural number. As an example, suppose that we are given a function from words of the English alphabet to graphs. Then we can think of a word in the English alphabet as a number written in base 27 with $a = 1$, $b = 2$ and so on. A graph on n nodes can be thought of as a sequence of $\binom{n}{2}$ binary symbols where each symbol corresponds to a potential edge and it is 1 iff the edge actually is there. For instance suppose that we are looking at graphs with 3 nodes, and hence the possible edges are $(1, 2), (1, 3)$ and $(2, 3)$. If the graph only contains the edges $(1, 3)$ and $(2, 3)$ we code it as 011. Add a leading 1 and consider the result as a number written in binary notation (our example corresponds to $(1011)_2 = 11$). It is easy to see that the mapping from graphs to numbers is easy to compute and easy to invert and thus we can use this representation of graphs as well as any other. Thus a function from words over the English alphabet to graphs can be represented as a function from natural numbers to natural numbers.

In a similar way one can see that most objects that have any reasonable formal representation can be represented as natural numbers. This fact will be used constantly throughout these notes.

After this detour let us return to the question of which functions are mechanically computable. Mechanically computable functions are often called recursive functions. The reason for this will soon be obvious.

2.1 Primitive Recursive Functions

The name "recursive" comes from the use of recursion, i.e. when a function value $f(x+1)$ is defined in terms of previous values $f(0), f(1) \dots f(x)$. The primitive recursive functions define a large class of computable functions which contains most natural functions. It contains some basic functions and then new primitive recursive functions can be built from previously defined primitive recursive functions either by composition or primitive recursion. Let us give a formal definition.

Definition 2.1 *The following functions are primitive recursive*

1. *The successor function,* $\sigma(x) = x+1$.
2. *Constants,* $m(x) = m$ *for any constant* m.
3. *The projections,* $\pi_i^n(x_1, x_2 \dots x_n) = x_i$ *for* $1 \leq i \leq n$ *and any* n.

   The primitive recursive functions are also closed under the following two operations. Assume that $g, h, g_1, g_2 \dots g_m$ *are known to be primitive recursive functions, then we can form new primitive recursive functions in the following ways.*

4. *Composition,* $f(x_1, x_2 \dots x_n) = h(g_1(x_1, \dots x_n), g_2(x_1, \dots, x_n), \dots g_m(x_1, \dots x_n))$
5. *Primitive recursion The function defined by*
   - $f(0, x_2, x_3, \dots x_n) = g(x_2, x_3, \dots x_n)$
   - $f(x_1+1, x_2, x_3, \dots x_n) = h(x_1, f(x_1, \dots x_n), x_2, \dots x_n)$

To get a feeling for this definition let us prove that some common functions are primitive recursive.

Example 2.2 Addition is defined as

$$
\begin{aligned}
Add(0, x_2) &= \pi_1^1(x_2) \\
Add(x_1+1, x_2) &= \sigma(\pi_2^3(x_1, Add(x_1, x_2), x_2)) \\
&= \sigma(Add(x_1, x_2))
\end{aligned}
$$

It will be very cumbersome to follow the notation of the definition of the primitive recursive functions strictly. Thus instead of the above, not very transparent (but formally correct definition) we will use the equivalent, more transparent (but formally incorrect) version stated below.

$$\begin{aligned} Add(0, x_2) &= x_2 \\ Add(x_1+1, x_2) &= Add(x_1, x_2) + 1 \end{aligned}$$

Example 2.3 Multiplication can be defined as

$$\begin{aligned} Mult(0, x_2) &= 0 \\ Mult(x_1+1, x_2) &= Add(x_2, Mult(x_1, x_2)) \end{aligned}$$

Example 2.4 We cannot define subtraction as usual since we require the answer to be nonnegative[1]. However, we can define a function which takes the same value as subtraction whenever it is positive and otherwise takes the value 0. First define a function on one variable which is basically subtraction by 1.

$$\begin{aligned} Sub1(0) &= 0 \\ Sub1(x+1) &= x \end{aligned}$$

and now we can let

$$\begin{aligned} Sub(x_1, 0) &= x_1 \\ Sub(x_1, x_2+1) &= Sub1(Sub(x_1, x_2)). \end{aligned}$$

Here for convenience we have interchanged the order of the arguments in the definition of the recursion but this can be justified by the composition rule.

Example 2.5 If $f(x, y) = \prod_{i=0}^{y-1} g(x, i)$ where we let $f(x, 0) = 1$ and g is primitive recursive then so is f since it can be defined by

$$\begin{aligned} f(x, 0) &= 1 \\ f(x, y+1) &= Mult(f(x, y), g(x, y)). \end{aligned}$$

Example 2.6 We can define a miniature version of the signum function by

$$\begin{aligned} Sg(0) &= 0 \\ Sg(x+1) &= 1 \end{aligned}$$

[1] This is due to the fact that we have decided to work with natural numbers. If we instead would be working with integers the situation would be different

and this allows us to define equality by

$$Eq(m,n) = Sub(1, Add(Sg(Sub(n,m)), Sg(Sub(m,n))))$$

since $Sub(n,m)$ and $Sub(m,n)$ are both zero iff $n = m$. Equality is here defined as by $Eq(m,n) = 1$ if m and n are equal and $Eq(m,n) = 0$ otherwise.

Equality is not really a function put a predicate of pairs of numbers i.e. a property of pairs of numbers. However, as we did above, it is convenient to identify predicates with functions that take the values 0 and 1, letting the value of the function be 1 exactly when the predicate is true. With this convention we define a predicate to be primitive recursive exactly when the corresponding function is primitive recursive. This naturally leads to an efficient way to prove that more functions are primitive recursive. Namely, let g and h be primitive recursive functions and let P be a primitive recursive predicate. Then the function $f(x)$ defined by $g(x)$ if $P(x)$ and $h(x)$ otherwise will be primitive recursive since it can be written as

$$Add(Mult(g(x), P(x)), Mult(h(x), Sub(1, P(x))))$$

(which in ordinary notation is $(P * g + (1 - P) * h)$.

Continuing along these lines it is not difficult (but tedious) to prove that most simple functions are primitive recursive. Let us now argue that all primitive recursive functions are mechanically computable. Of course this can only be an informal argument since "mechanically computable" is only an intuitive notion.

Each primitive recursive function is defined as a sequence of statements starting with basic functions of the types 1-3 and then using rules 4-5. We will call this a *derivation* of the function. We will argue that primitive recursive functions are mechanically computable by induction over the complexity of the derivation (i.e. the number of steps in the derivation).

The simplest functions are the basic functions 1-3 and, arguing informally, are easy to compute. In general a primitive recursive function f will be obtained using the rules 4 and 5 from functions defined previously. Since the derivations of these functions are subderivations of the given derivation, we can conclude that the functions used in the definition are mechanically computable. Suppose the new function is constructed by composition, then we can compute f by first computing the g_i and then computing h of the results. On the other hand if we use primitive recursion then we can compute f when the first argument is 0 since it then agrees with g which is computable by induction and then we can see that we can compute f in general by induction over the size of the first argument. This finishes the informal argument that all primitive recursive functions are mechanically computable.

Before we continue, let us note the following: If we look at the proof in the case of multiplication it shows that multiplication is mechanically computable but it gives an extremely inefficient algorithm. Thus the present argument has nothing to do with computing efficiently.

Although we have seen that most simple functions are primitive recursive there are in fact functions which are mechanically computable but are not primitive recursive. We will give one such function which, we have to admit, would not be the first one would like to compute but which certainly is very important from a theoretical point of view.

A derivation of a primitive recursive function is just a finite number of symbols and thus we can code it as a number. If the coding is reasonable it is mechanically computable to decide, given a number, whether the number corresponds to a correct derivation of a primitive recursive function in one variable. Now let f_1 be the primitive recursive function in one variable which corresponds to the smallest number giving such a legal derivation and then let f_2 be the function which corresponds to the second smallest number and so on. Observe that given x it is possible to mechanically find the derivation of f_x by the following mechanical but inefficient procedure. Start with 0 and check the numbers in increasing order whether they correspond to correct derivations of a function in one variable. The x'th legal derivation found is the derivation of f_x. Now let

$$V(x) = f_x(x) + 1.$$

By the above discussion V is mechanically computable, since once we have found the derivation of f_x we can compute it on any input. On the other hand we claim that V does not agree with any primitive recursive function. If V was primitive recursive then $V = f_y$ for some number y. Now look at the value of V at the point y. By the definition of V the value should be $f_y(y) + 1$. On the other hand if $V = f_y$ then it is $f_y(y)$. We have reached a contradiction and we have thus proved:

Theorem 2.7 *There are mechanically computable functions which are not primitive recursive.*

The method of proof used to prove this theorem is called *diagonalization*. To see the reason for this name think of an infinite two-dimensional array with natural numbers along one axis and the primitive recursive functions on the other. At position (i, j) we write the number $f_j(i)$. We then construct a function which is not primitive recursive by going down the diagonal and making sure that our function disagrees with f_i on input i. The idea is similar to the proof that Cantor used to prove that the real numbers are not denumerable.

The above proof demonstrates something very important. If we want to have a characterization of all mechanically computable functions the description cannot be mechanically computable by itself. By this we mean that given x we should not be able to find f_x in a mechanical way. If we could find f_x then the above defined function V would be mechanically computable and we would get a function which was not in our list.

2.2 Partial recursive functions

The way around the problem mentioned last in the last section is to allow a derivation to define a function which is only *partial* i.e. is not defined for all inputs. We will do this by giving another way of forming new function. This modification will give a new class of functions called the *partial recursive functions*.

Definition 2.8 *The partial recursive functions contains the basic functions defined by 1-3 for primitive recursive functions and are closed under the operations 4 and 5. There is an extra way of forming new functions:*

6. *Unbounded search Assume that g is a partial recursive function and let $f(x_1, \ldots x_n)$ be the least m such that $g(m, x_1, \ldots, x_n) = 0$ and such that $g(y, x_1, \ldots, x_n)$ is defined for all $y < m$. If no such m exists then $f(x_1, \ldots, x_n)$ is undefined. Then f is partial recursive.*

Our first candidate for the class of mechanically computable functions will be a subclass of the partial recursive functions.

Definition 2.9 *A function is recursive (or total recursive) if it is a partial recursive function which is total, i.e. which is defined for all inputs.*

Observe that a recursive function is in an intuitive sense mechanically computable. To see this we just have to check that the property of mechanical computability is closed under the rule 6, given that f is defined. But

Figure 1: A Turing machine

this follows since we just have to keep computing g until we find a value for which it takes the value 0. The key point here is that since f is total we know that eventually there is going to be such a value.

Also observe that there is no obvious way to determine whether a given derivation defines a total function and thus defines a recursive function. The problem being that it is difficult to decide whether the defined function is total (i.e. if for each value of $x_1, x_2, \ldots x_n$ there is an m such that $g(m, x_1, x_2, \ldots x_n) = 0$. This implies that we will not be able to imitate the proof of Theorem 2.7 and thus there is some hope that this definition will give all mechanically computable functions. Let us next describe another approach to define mechanically computable functions.

2.3 Turing Machines

The definition of mechanically computable functions as recursive functions given in the last section is due to Kleene. Other definitions of mechanically computable were given by Church (effective calculability, also by equations), Post (canonical systems, as rewriting systems) and Turing (Turing machines, a type of primitive computer). Of these we will only look closer at Turing machines. This is probably the definition which to most of us today, after the invention of the modern computer, seems most natural.

A Turing machine is a very primitive computer. A simple picture of one is given in Figure 1. The infinite *tape* serves as memory and input and output device. Each square can contain one symbol from a finite alphabet which we will denote by Σ. It is not important which alphabet the machine uses and thus let us think of it as $\{0, 1, B\}$ where B symbolizes the blank square. The input is initially given on the tape. At each point in time the *head* is located at one of the tape squares and is in one of a finite number of *states*. The machine reads the content of the square the head is located at, and

State	Symbol	New State	New Symbol	Move
q_0	0	q_1	B	R
q_0	1	q_0	B	R
q_0	B	q_h	1	
q_1	0,1	q_1	B	R
q_1	B	q_h	0	

Table 1: The next step function of a simple Turing machine

based on this value and its state, it writes something into the square, enters a potentially new state and moves left or right. Formally this is described by the *next-move* function

$$f : Q \times \Sigma \rightarrow Q \times \Sigma \times \{R, L\}$$

where Q is the set of possible states and R(L) symbolizes moving right (left). From an intuitive point of view the next-move function is the program of the machine.

Initially the machine is in a special start-state, q_0, and the head is located on the leftmost square of the input. The tape squares that do not contain any part of the input contain the symbol B. There is a special halt-state, q_h, and when the machine reaches this state it halts. The output is now defined by the non-blank symbols on the tape.

It is possible to make the Turing machine more efficient by allowing more than one tape. In such a case there is one head on each tape. If there are k tapes then the next-step function depends on the contents of all k squares where the heads are located, it describes the movements of all k heads and what new symbols to write into the k squares. If we have several tapes then it is common to have one tape on which the input is located, and not to allow the machine to write on this tape. In a similar spirit there is one output-tape which the machine cannot read. This convention separates out the tasks of reading the input and writing the output and thus we can concentrate on the heart of the matter, the computation.

However, most of the time we will assume that we have a one-tape Turing machine. When we are discussing computability this will not matter, but later when considering efficiency of computation results will change slightly.

Example 2.10 Let us define a Turing Machine which checks if the input contains only ones and no zeros. It is given in Table 1.

Thus the machine starts in state q_0 and remains in this state until it has seen a "0". If it sees a "B" before it sees a "0" it accepts. If it ever sees a "0" it erases the rest of the input, prints the answer 0 and then halts.

Example 2.11 Programming Turing machines gets slightly cumbersome and as an example let us give a Turing machine which computes the sum of two binary numbers. We assume that we are given two numbers with least significant bit first and that there is a B between the two numbers. To make things simpler we also assume that we have a special output-tape on which we print the answer, also here beginning with the least significant bit.

To make the representation compact we will let the states have two indices, The first index is just a string of letters while the other is a number, which in general will be in the range 0 to 3. Let division be integer division and let $lsb(i)$ be the least significant bit of i. The program is given in Table 2.3, where we assume for notational convenience that the machine starts in state $q_{0,0}$:

It will be quite time-consuming to explicitly give Turing machines which compute more complicated functions. For this reason this will be the last Turing machine that we specify explicitly. To be honest there are more economic ways to specify Turing machines. One can build up an arsenal of small machines doing basic operations and then define composition of Turing machines. However, since programming Turing machines is not our main task we will not pursue this direction either.

A Turing machine defines only a partial function since it is not clear that the machine will halt for all inputs. But whenever a Turing machine halts for all inputs it corresponds to a total function and we will call such a function *Turing computable.*

The "Turing computable functions" is a reasonable definition of the mechanically computable functions and thus the first interesting question is how this new class of functions relates to the recursive functions. We have the following theorem.

Theorem 2.12 *A function is Turing computable iff it is recursive.*

We will not give the proof of this theorem. The proof is rather tedious, and hence we will only give an outline of the general approach. The easier part of the theorem is to prove that if a function is recursive then it is Turing computable. Before, when we argued that recursive functions were mechanically computable, most people who have programmed a modern

State	Symbol	New State	New Symbol	Move	Output
$q_{0,i}$	$0,1(=j)$	$q_{x,i+j}$	B	R	
$q_{x,i}$	$0,1$	$q_{xm,i}$	same	R	
$q_{x,i}$	B	$q_{xo,i}$	B	R	
$q_{xo,i}$	B	$q_{xo,i}$	B	R	
$q_{xo,i}$	$0,1(=j)$	$q_{yc,\frac{i+j}{2}}$	B	R	lsb(i+j)
$q_{yc,i}$	$0,1(=j)$	$q_{yc,\frac{i+j}{2}}$	B	R	lsb(i+j)
$q_{yc,i}$	B	q_h	B		i
$q_{xm,i}$	$0,1$	$q_{xm,i}$	same	R	
$q_{xm,i}$	B	$q_{sy,i}$	B	R	
$q_{sy,i}$	B	$q_{sy,i}$	B	R	
$q_{sy,i}$	$0,1(=j)$	$q_{y,\frac{i+j}{2}}$	B	R	lsb(i+j)
$q_{y,i}$	$0,1$	$q_{sx,i}$	same	L	
$q_{y,i}$	B	$q_{yo,i}$	B	L	
$q_{sx,i}$	B	$q_{sx,i}$	B	L	
$q_{sx,i}$	$0,1$	$q_{fx,i}$	same	L	
$q_{fx,i}$	$0,1$	$q_{fx,i}$	same	L	
$q_{fx,i}$	B	$q_{0,i}$	B	R	
$q_{yo,i}$	B	$q_{yo,i}$	B	L	
$q_{yo,i}$	$0,1$	$q_{xf,i}$	same	L	
$q_{xf,i}$	$0,1$	$q_{xf,i}$	same	L	
$q_{xf,i}$	B	$q_{cx,i}$	B	R	
$q_{cx,i}$	$0,1(=j)$	$q_{cx,\frac{i+j}{2}}$	B	R	lsb(i+j)
$q_{cx,i}$	B	q_h	B		i

Table 2: A Turing machine for addition

computer probably felt that without too much trouble one could write a program that would compute a recursive function. It is harder to program Turing machines, but still feasible.

For the other implication one has to show that any Turing computable function is recursive. The way to do this is to mimic the behavior of the Turing machine by equations. This gets fairly involved and we will not describe this procedure here.

2.4 Church's thesis

In the last section we stated the theorem that recursive functions are identical to the Turing computable functions. It turns out that all the other attempts to formalize mechanically computable functions give the same class of functions. This leads one to believe that we have captured the right notion of computability and this belief is usually referred to as Church's thesis. Let us state it for future reference.

CHURCH'S THESIS: *The class of recursive functions is the class of mechanically computable functions, and any reasonable definition of mechanically computable will give the same class of functions.*

Observe that Church's thesis is not a mathematical theorem but a statement of experience. Thus we can use such imprecise words as "reasonable". Church's thesis is very convenient to use when arguing about computability. Since any high level computer language describes a reasonable model of computation the class of functions computable by high level programs is included in the class of recursive functions. Thus as long as our descriptions of procedures are detailed enough so that we feel certain that we could write a high level program to do the computation, we can draw the conclusion that we can do the computation on a Turing machine or by a recursive function. In this way we do not have to worry about actually programming the Turing machine.

For the remainder of these notes we will use the term "recursive functions" for the class of functions described by Church's thesis. Sometimes, instead of saying that a given function, f, is a recursive function we will phrase this as "f is computable". When we argue about such functions we will usually argue in terms of Turing machines but the algorithms we describe will only be specified quite informally.

2.5 Functions, sets and languages

If a function f only takes two values (which we assume without loss of generality to be 0 and 1) then we can identify f with the set, A, of inputs for which the function takes the value 1. In formulas

$$x \in A \Leftrightarrow f(x) = 1.$$

In this connection sets are also called *languages*, e.g. the set of prime numbers could be called the language of prime numbers. The reason for this is historical and comes from the theory of formal languages. The function f is called the *characteristic* function of A. Sometimes the characteristic function of A will be denoted by χ_A. A set is called *recursive* iff its characteristic function is recursive. Thus A is recursive iff given x one can mechanically decide whether $x \in A$.

2.6 Recursively enumerable sets

We have defined recursive sets to be the sets for which membership can be tested mechanically i.e. a set A is recursive if given x it is computable to test whether $x \in A$. Another interesting class of sets is the class of sets which can be listed mechanically.

Definition 2.13 *A set A is recursively enumerable iff there is a Turing machine M_A which, when started on the empty input tape, lists the members of A on its output tape.*

It is important to remember that, while any member of A will eventually be listed, the members of A are not necessarily listed in order and that M will probably never halt since A is infinite most of the time. Thus if we want to know whether $x \in A$ it is not clear how to use M for this purpose. We can watch the output of M and if x appears we know that $x \in A$, but if we have not seen x we do not know whether $x \notin A$ or we have not waited long enough. If we would require that A was listed in order we could check whether $x \in A$ since we would only have had to wait until we had seen x or a number greater than x.[2] Thus in this case we can conclude that A is recursive, but in general this is not true.

[2] There is a slightly subtle point here since it might be the case that M never outputs such a number, which would happen in the case when A is finite and does not contain x or any larger number. However also in this case A is recursive since any finite set is recursive. It is interesting to note that given the machine M it is not clear which alternative should be used to recognize A, but one of them will work and that is all we care about.

Theorem 2.14 *If a set is recursive then it is recursively enumerable. However there are sets that are recursively enumerable that are not recursive.*

Proof: That recursive implies recursively enumerable is not too hard, the procedure below will even print the members of A in order.

For $i = 0, 1 \dots \infty$
    If $i \in A$ print i.

Since it is computable to determine whether $i \in A$ this will give a correct enumeration of A.

The other part of the theorem is harder and requires some more notation. A Turing machine is essentially defined by the next-step function which can be described by a number of symbols and thus can be coded as an integer. Let us outline in more detail how this is done. We have described a Turing machine by a number of lines where each line contains the following items: State, Symbol, New state, New Symbol, Move and Output. Let us make precise how to code this information. A state should be written as q_x where x is a natural number written in binary. A symbol is from the set $\{0, 1, B\}$, while a move is either R or L and the output is either $0, 1$ or B. Each item is separated from the next by the special symbol &, the end of a line is marked as & & and the end of the specification is marked as & & &. We assume that the start state is always q_0 and the halt state q_1. With these conventions a Turing machine is completely specified by a finite string over the alphabet $\{0, 1, B, \&, R, L, q\}$. This coding is also efficient in the sense that given a string over this alphabet it is possible to mechanically decide whether it is a correct description of a Turing machine (think about this for a while). By standard coding we can think of this finite string as a number written in base 8. Thus we can uniquely code a Turing machine as a natural number.

For technical reason we allow the end of the specification not to be the last symbols in the coding. If we encounter the end of the specification we will just discard the rest of the description. This definition implies that each Turing machine occurs infinitely many times in any natural enumeration.

We will denote the Turing machine which is given by the description corresponding to y by M_y. We again emphasize that given y it is possible to mechanically determine whether it corresponds to a Turing machine and in such a case find that Turing machine. Furthermore we claim that once we have the description of the Turing machine we can run it on any input (*simulate* M_y on a given input). We make this explicit by stating a theorem we will not prove.

Theorem 2.15 *There is a* universal Turing machine *which on input* (x, y, z) *simulates* z *computational steps of* M_y *on input* x. *By this we mean that if* M_y *halts with output* w *on input* x *within* z *steps then also the universal machine outputs* w. *If* M_y *does not halt within* z *steps then the universal machine gives output "not halted". If* y *is not the description of a legal Turing machine, the universal Turing machine enters a special state* q_{ill}, *where it usually would halt, but this can be modified at will.*

We will sometimes allow z to take the value ∞. In such a case the universal machine will simulate M_y until it halts or go on for ever without halting if M_y does not halt on input x. The output will again agree with that of M_y.

In a more modern language, the universal Turing machine is more or less an interpreter since it takes as input a Turing machine program together with an input and then runs the program. We encourage the interested reader to at least make a rough sketch of a program in his favorite programming language which does the same thing as the universal Turing machine.

We now define a function which is in the same spirit of the function V which we proved not to be primitive recursive. To distinguish it we call it V_T.

$$V_T(x) = \begin{cases} 1, & \text{if } M_x \text{ halts on input } x \text{ with output } 0; \\ 0, & \text{otherwise.} \end{cases}$$

V_T is the characteristic function of a set which we will denote by K_D. We call this set "the diagonal halting set" since it is the set of Turing machines which halt with output 0 when given their own encoding as input. We claim that K_D is recursively enumerable but not recursive. To prove the first claim observe that K_D can be enumerated by the following procedure

For $i = 1, 2 \ldots \infty$
    For $j = 1, 2, \ldots i$, If M_j is legal, run M_j, i steps on input j, if it halts within these i steps and gives output 0 and we have not listed j before, print j.

Observe that this is an recursive procedure using the universal Turing machine. The only detail to check is that we can decide whether j has been listed before. The easiest way to do this is to observe that j has not been listed before precisely if $j = i$ or M_j halted in exactly i steps. The procedure lists K_D since all numbers ever printed are by definition members in K_D and if $x \in K_D$ and M_x halts in T steps on input x then x will be listed for $i = \max(x, T)$ and $j = x$.

To see that K_D is not recursive, suppose that V_T can be computed by a Turing machine M. We know that $M = M_y$ for some y. Consider what happens when M is fed input y. If it halts with output 0 then $V_T(y) = 1$. On the other hand if M does not halt with output 0 then $V_T(y) = 0$. In either case M_y makes an error and hence we have reached a contradiction. This finishes the proof of Theorem 2.14 ∎

We have proved slightly more than was required by the theorem. We have given an explicit function which cannot be computed by a Turing machine. Let us state this as a separate theorem.

Theorem 2.16 *The function V_T cannot be computed by a Turing machine, and hence is not recursive.*

2.7 Some facts about recursively enumerable sets

Recursion theory is really the predecessor of complexity theory and let us therefore prove some of the standard theorems to give us something to compare with later. In this section we will abbreviate recursively enumerable as "r.e.".

Theorem 2.17 *A is recursive if and only if both A and the complement of A, ($\bar{A}$) are r.e.*

Proof: If A is recursive then also $\bar{A}$ is recursive (we get a machine recognizing $\bar{A}$ from a machine recognizing A by changing the output). Since any recursive set is r.e. we have proved one direction of the theorem. For the converse, to decide whether $x \in A$ we just enumerate A and $\bar{A}$ in parallel, and when x appears in one of lists, which we know it will, we can give the answer and halt. ∎

From Theorem 2.16 we have the following immediate corollary.

Corollary 2.18 *The complement of K_D is not r.e..*

For the next theorem we need the fact that we can code pairs of natural numbers as natural numbers. For instance one such coding is given by $f(x, y) = (x + y)(x + y + 1)/2 + x$.

Theorem 2.19 *A is r.e. iff there is a recursive set B such that $x \in A \Leftrightarrow \exists y\ (x, y) \in B$.*

Proof: If there is such a B then A can be enumerated by the following program:

For $z = 0, 1, 2, \ldots \infty$
  For $x = 0, 1, 2 \ldots z$ If for some $y \leq z$ we have $(x, y) \in B$ and $(x, y') \notin B$ for $y' < y$ and x has not been printed before then print x.

First observe that x has not been printed before if either x or y is equal to z. By the relation between A and B this program will list only members of A and if $x \in A$ and y is the smallest number such that $(x, y) \in B$ then x is listed for $z = \max(x, y)$.

To see the converse, let M_A be the Turing machine which enumerates A. Define B to be the set of pairs (x, y) such that x is output by M_A in at most y steps. By the existence of the universal Turing machine it follows that B is recursive and by definition $\exists y(x, y) \in B$ precisely when x appears in the output of M_A, i.e. when $x \in A$. This finishes the proof of Theorem 2.19. ∎

The last theorem says that r.e. sets are just recursive sets plus an existential quantifier. We will later see that there is a similar relationship between the complexity classes P and NP.

Let the halting set, K, be defined by

$$K = \{(x, y) | M_y \text{ is legal and halts on input } x\}.$$

To determine whether a given pair $(x, y) \in K$ is for natural reasons called the *halting problem*. This is closely related to the diagonal halting problem which we have already proved not to be recursive in the last section. Intuitively this should imply that the halting problem also is not recursive and in fact this is the case.

Theorem 2.20 *The halting problem is not recursive.*

Proof: Suppose K is recursive i.e. that there is a Turing machine M which on input (x, y) gives output 1 precisely when M_y is legal and halts on input x. We will use this machine to construct a machine that computes V_T using M as a subroutine. Since we have already proved that no machine can compute V_T this will prove the theorem.

Now consider an input x and that we want to compute $V_T(x)$. First decide whether M_x is a legal Turing machine. If it is not we output 0 and halt. If M_x is a legal machine we feed the pair (x, x) to M. If M outputs 0 we can safely output 0 since we know that M_x does not halt on input x. On the other hand if M outputs 1 we use the universal machine on input (x, x, ∞) to determine the output of M_x on input x. If the output is 0 we give the answer 1 and otherwise we answer 0. This gives a mechanical procedure that computes V_T and we have reached the desired contradiction. ∎

It is now clear that other problems can be proved to be non-recursive by a similar technique. Namely we assume that the given problem is recursive and we then make an algorithm for computing something that we already know is not recursive. One general such method is by a standard type of reduction and let us next define this concept.

Definition 2.21 *For sets A and B let the notation $A \leq_m B$ mean that there is a recursive function f such that $x \in A \Leftrightarrow f(x) \in B$.*

The reason for the letter m on the less than sign is that one usually defines several different reductions. This particular reduction is usually referred to as a many-one reduction. We will not study other definitions in detail, but since the only reduction we have done so far was not a many-one reduction but a more general notion called Turing reduction, we will define also this reduction.

Definition 2.22 *For sets A and B let the notation $A \leq_T B$ mean that given a Turing machine that recognizes B then using this machine as a subroutine we can construct a Turing machine that recognizes A.*

The intuition for either of the above definitions is that A is not harder to recognize than B. This is formalized as follows:

Theorem 2.23 *If $A \leq_m B$ and B is recursive then A is recursive.*

Proof: To decide whether $x \in A$, first compute $f(x)$ and then check whether $f(x) \in B$. Since both f and B are recursive this is a recursive procedure and it gives the correct answer by the definition of $A \leq_m B$. ∎

Clearly the similar theorem with Turing reducibility rather than many-one reducibility is also true (prove it). However in the future we will only reason about many-one reducibility. Next let us define the hardest problem within a given class.

Definition 2.24 *A set A is r.e.-complete iff*

1. *A is r.e.*
2. *If B is r.e. then $B \leq_m A$.*

We have

Theorem 2.25 *The halting set is r.e.-complete.*

Proof: The fact that the halting problem is r.e. can be seen in a similar way that the diagonal halting problem K_D was seen to be r.e.. Just run more and more machines more and more steps and output all pairs of machines and inputs that leads to halting.

To see that it is complete we have to prove that any other r.e. set, B can be reduced to K. Let M be the Turing machine that enumerates B. Define M' to be the Turing machine which on input x runs M until it outputs x (if ever) and then halts with output 0. Then M' halts precisely when $x \in B$. Thus if $M' = M_y$ we can let $f(x) = (x, y)$ and this will give a reduction from B to K. The proof is complete. ∎

It is also true that the diagonal halting problem is r.e.-complete, but we omit the proof. There are many other (often more natural) problems that can be proved r.e.-complete (or to be even harder) and let us define two such problems.

The first problem is called *tiling* a can be thought of as a two-dimensional domino game. Given a finite set of squares (which will be called *tiles*), each with a marking on all four sides and one tile placed at the origin in the plane. The question is whether it is possible to cover the entire positive quadrant with tiles such that on any two neighboring tiles, the markings agree on their common side and such that each tile is equal to one of the given tiles.

Theorem 2.26 *The complement problem of tiling is r.e.-complete.*

Proof: (Outline) Given a Turing machine M_x we will construct a set of tiles and a tile at the origin such that the entire positive quadrant can be tiled iff M_x does not halt on the empty input. The problem whether a Turing machine halts on the empty input is not recursive (this is one of the exercises in the end of this chapter). We will construct the tiles in such a way that the only way to put down tiles correctly will be to make them describe a computation of M_x. The tile at the origin will make sure that the machine starts correctly (with some more complication this tile could have been eliminated also).

Let the *state* of a tape cell be the content of the cell with the additional information whether the head is there and in such a case which state the machine is in. Now each tile will describe the state of three adjacent cells. The tile to be placed at position (i, j) will describe the state of cells j, $j + 1$ and $j + 2$ at time i of the computation. Observe that this implies that tiles which are to the left and right of each other will describe overlapping parts of the tape. However, we will make sure that the descriptions do not conflict.

A tile will thus be partly be specified by three cell-states s_1, s_2 and s_3 (we call this the *signature* of the tile) and we need to specify how to mark its four sides. The left hand side will be marked by (s_1, s_2) and the right hand side by (s_2, s_3). Observe that this makes sure that there is no conflict in the descriptions of a cell by different tiles. The markings on the top and the bottom will make sure that the computation proceeds correctly.

Suppose that the states of cells j, $j + 1$, and $j + 2$ are s_1, s_2, and s_3 at time t. Consider the states of these cells at time $t + 1$. If one of the s_i tells us that the head is present we know exactly what states the cells will be in. On the other hand if the head is not present in any of the three cells there might be several possibilities since the head could be in cells $j - 1$ or $j + 3$ and move into one of our positions. In a similar way there might be one or many (or even none) possible states for the three cells at time $t - 1$. For each possibility $(s_1^{-1}, s_2^{-1}, s_3^{-1})$ and $(s_1^{+1}, s_2^{+1}, s_3^{+1})$ of states in the previous and next step we make a tile. The marking on the lower side is $(s_1^{-1}, s_2^{-1}, s_3^{-1}), (s_1, s_2, s_3)$ while the marking on the top side is $(s_1, s_2, s_3), (s_1^{+1}, s_2^{+1}, s_3^{+1})$. This completes the description of the tiles.

Finally at the origin we place a tile which describes that the machine starts in the first cell in state q_0 and blank tape. Now it is easy to see that a valid tiling describes a computation of M_x and the entire quadrant can be tiled iff M_x goes on for ever i.e. it does not halt.

There are a couple of details to take care of. Namely that new heads don't enter from the left and that the entire tape is blank from the beginning.

A couple of special markings will take care of this. We leave the details to the reader. ∎

The second problem we will consider is number theoretic statements, i.e. given a number theoretic statement is it false or true? One particular statement people have been interested in for a long time (which supposedly was proved true in 1993) is Fermat's last theorem, which can be written as follows

$$\forall n > 2\ \forall x, y, z x^n + y^n = z^n \leftarrow xyz = 0.$$

In general a number theoretic statement involves the quantifiers $\forall$ and $\exists$, variables and usual arithmetical operations. Quantifiers range over natural numbers.

Theorem 2.27 *The set of true number theoretic statements is not recursive.*

Remark 2.28 *In fact the set of true number theoretic statements is not even r.e. but have a much more complicated structure. To prove this would lead us to far into recursion theory. The interested reader can consult any standard text in recursion theory.*

Proof: (Outline) Again we will prove that we can reduce the halting problem to the given problem. This time we will let an enourmous integer z code the computation. Thus assume we are given a Turing machine M_x and that we want to decide whether it halts on the empty input.

The state of each cell will be given by a certain number of bits in the binary expansion of z. Suppose that each cell has at most $S \leq 2^r$ states. A computation of M_x that runs in time t never uses more than t tape cells and thus such a computation can be described by the content of t^2 cells (i.e. t cells each at t different points in time). This can now be coded as rt^2 bits and these bits concatenated will be the integer z. Now let A_x be an arithmetic formula such that $A_x(z, t)$ is true iff z is a rt^2 bit integer which describes a correct computation for M_x which have halted. To check that such a formula exists requires a fair amount of detailed reasoning and let us just sketch how to construct it. First one makes a predicate $Cell(i, j, z, t, p)$ which is true iff p is the integer that describes the content of cell i at time j. This amounts to extracting the r bits of z which are in position starting at $(it + j)r$. Next one makes a predicate $Move(p_1, p_2, p_3, q)$ which says that if p_1, p_2 and p_3 are the states of squares $i-1$, i and $i+1$ at time j then q is the resulting state of square i at time $j+1$. The $Cell$ predicate is from an intuitive point of view very arithmetic (and thus we hope the reader feels that it can be constructed). $Move$ on the other hand is of constant size (there are only 2^{4r} inputs, which is a constant depending only on x and independent of t) and thus can be coded by brute force. The predicate $A_x(z,t)$ is now equivalent to the conjunction of

$$\forall i,j,p_1,p_2,p_3,q\ Cell(i-1,j,z,t,p_1)\wedge$$
$$Cell(i,j,z,t,p_2)\wedge Cell(i+1,j,z,t,p_3)\wedge$$
$$Cell(i,j+1,z,t,p_q)\Rightarrow Move(p_1,p_2,p_3,q)$$

and

$$\forall q'\ Cell(1,t,z,t,q')\Rightarrow Stop(q')$$

where $Stop(p)$ is true if p is a haltstate. Now we are almost done since M_x halts iff

$$\exists z,tA_x(z,t)$$

and thus if we can decide the truth of arithmetic formulae with quantifiers we can decide if a given Turing machine halts. Since we know that this is not possible we have finished the outline of the proof. ∎

Remark 2.29 *It is interesting to note that (at least to me) the proofs of the last two theorems are in some sense counter intuitive. It seems like the hard part of the tiling problem is what to do at points where we can put down many different tiles (we never know if we made the correct decision). This is not utilized in the proof. Rather at each point we have only one choice and the hard part is to decide whether we can continue for ever. A similar statement is true about the other proof.*

Let us explicitly state a theorem we have used a couple of times.

Theorem 2.30 *If A is r.e.-complete then A is not recursive.*

Proof: Let B be a set that is r.e. but not recursive (e.g. the halting problem) then by the second property of being r.e.-complete $B\leq_m A$. Now if A was recursive then by Theorem 2.7.6 we could conclude that B is recursive, contradicting the initial assumption that B is not recursive. ∎

Before we end this section let us make an informal remark. What does it mean that the halting problem is not recursive? Experience shows that for most programs that do not halt there is a simple reason that they do not halt. They often tend to go into an infinite loop and of course such things can be detected. We have only proved that there is not a single program which when given as input the description of a Turing machine and an input to that machine, the program will *always* give the correct answer to the question whether the machine halts or not. One final definition: A problem that is not recursive is called *undecidable*. Thus the halting problem is undecidable.

2.8 Gödel's incompleteness theorem

Since we have done many of the pieces let us briefly outline a proof of Gödel's incompleteness theorem. This theorem basically says that there are statements in arithmetic which neither have proof or a disproof. We want to avoid a too elaborate machinery and hence we will be rather informal and give an argument in the simplest case. However, before we state the theorem we need to address what we mean by "statement in arithmetic" and "proof".

Statements in arithmetic will simply be the formulas considered in the last examples, i.e. quantified formulas where the variables values which are natural numbers. We encourage the reader to write common theorems and conjectures in number theory in this form to check its power.

The notion of a proof is more complicated. One starts with a set of *axioms* and then one is allowed to combine axioms (according to some rules) to derive new theorems. A proof is then just such a derivation which ends with the desired statement.

First note that most proofs used in modern mathematics is much more informal and given in a natural language. However, proof can be formalized (although most humans prefer informal proofs).

The most common set of axioms for number theory was proposed by Peano, but one could think of other sets of axioms. We call a set of axioms together with the rules how they can be combined a *proofsystem*. There are two crucial properties to look for in a proofsystem. We want to be able to prove all true theorem (this is called completeness) and we do not want to be able to prove any false theorems (this is called that the system is consistent). In particular, for each statement A we want to be able to prove exactly one of A and $\neg A$.

Our goal is to prove that there is no proof system that is both consistent and complete. Unfortunately, this is not true since we can as axioms take all true statements and then we need no rules for deriving new theorems. This is not a very practical proofsystem since there is no way to tell whether a given statement is indeed an axiom. Clearly the axioms need to be specified in a more efficient manner. We take the following definition.

Definition 2.31 *A proofsystem is recursive iff the set of proofs (and hence the set of axioms) form a recursive set.*

We can now state the theorem.

Theorem 2.32 *(Gödel) There is no recursive proofsystem which is both consistent and complete.*

Proof: Assume that there was indeed such a proofsystem. Then we claim that also the set of all theorems would be recursive. Namely to decide whether a statement A is true we could proceed as follows:

For $z = 0, 1, 2, \ldots \infty$
  If z is a correct proof of A output "true" and halt.
  If z is a correct proof of $\neg A$ output "false" and halt.

To check whether a given string is a correct proof is recursive by assumption and since the proofsystem is consistent and complete sooner or later there will be a proof of either A or $\neg A$. Thus this procedure always halts with the correct answer. However, by Theorem 2.27 the set of true statements is not recursive and hence we have reached a contradiction. ∎

2.9 Exercises

Let us end this section with a couple of exercises (with answers). The reader is encouraged to solve the exercises without looking too much at the answers.

II.1: Given x is it recursive to decide whether M_x halts on an empty input?

II.2: Is there any fixed machine M, such that given y, deciding whether M halts on input y is recursive?

II.3: Is there any fixed machine M, such that given y, deciding whether M halts on input y is not recursive?

II.4: Is it true that for each machine M, that given y, it is recursive to decide whether M halts on input y in y^2 steps?

II.5: Given x is it recursive to decide whether there exists a y such that M_x halts on y?

II.6: Given x is it recursive to decide whether for all y, M_x halts on y?

II.7: If M_x halts on empty input let $f(x)$ be the number of steps it needs before it halts and otherwise set $f(x) = 0$. Define the maximum time function by $MT(y) = \max_{x \leq y} f(x)$. Is the maximum time function computable?

II.8 Prove that the maximum time function (cf ex. II.7) grows at least as fast as any recursive function. To be more precise let g be any recursive function, then there is an x such that $MT(x) > g(x)$.

II.9 Given a set of rewriting rules over a finite alphabet and a starting string and a target string, is it decidable whether we, using the rewriting rules, can transform the starting string to the target string? An example of this instance is: Rewriting rules $ab \to ba$, $aa \to bab$ and $bb \to a$. Is it possible to transform $ababba$ to $aaaabbb$?

II.10 Given a set of rewriting rules over a finite alphabet and a starting string. Is it decidable whether we, using the rewriting rules, can transform the starting string to an arbitrarily long string?

II.11 Given a set of rewriting rules over a finite alphabet and a starting string. Is it decidable whether we, using the rewriting rules, can transform the starting string to an arbitrarily long string, if we restrict the left hand side of each rewriting rule to be of length 1?

2.10 Answers to exercises

II.1 The problem is undecidable. We will prove that if we could decide whether M_x halts on the empty input, then we could decide whether M_z halts on input y for an arbitrary pair z, y. Namely given z and y we make a machine M_x which basically looks like M_z but has a few special states. We have one special state for each symbol of y. On empty input M_x first goes trough all its special states which writes y on the tape. The machine then returns to the beginning of the tape and from this point on it behaves as M_z. This new machine halts on empty input-tape iff M_z halted on input y and thus if we could decide the former we could decide the latter which is known undecidable. To conclude the proof we only have to observe that it is recursive to compute the number x from the pair y and z.

II.2 There are plenty of machines of this type. For instance let M be the machine that halts without looking at the input (or any machine defining a total function). In one of these cases the set of y's for which the machine halts is everything which certainly is a decidable set.

II.3 Let M be the universal machine. Then M halts on input (x, y) iff M_x halts on input y. Since the latter problem is undecidable so is the former.

II.4 This problem is decidable by the existence of the universal machine. If we are less formal we could just say that running a machine a given number of steps is easy. What makes halting problems difficult is that we do not know for how many steps to run the machine.

II.5 Undecidable. Suppose we could decide this problem, then we show that we could determine whether a machine M_x halts on empty input. Given M_x we create a machine M_z which first erases the input and then behaves as M_x. We claim that M_z halts on some input iff M_x halts on empty input. Also it is true that we can compute z from x. Thus if we could decide whether M_z halts on some input then we could decide whether M_x halts on empty input, but this is undecidable by exercise II.1.

II.6 Undecidable. The argument is the same as in the previous exercise. The constructed machine M_z halts on all inputs iff it halts on some input.

II.7 MT is not computable. Suppose it was, then we could decide whether M_x halts on empty input as follows: First compute $MT(x)$ and then run M_x for $MT(x)$ steps on the empty input. If it halts in this number of steps, we know the answer and if it did not halt, we know by the definition of MT that it will never halt. Thus we always give the correct answer. However we know by exercise II.1 that the halting problem on empty input is undecidable. The contradiction must come from our assumption that MT is computable.

II.8 Suppose we had a recursive function g such that $g(x) \geq MT(x)$ for all x. Then $g(x)$ would work in the place of $MT(x)$ in the proof of exercise II.7 (we would run more steps than we needed to, but we would always get the correct answer). Thus there can be no such function.

II.9 The problem is undecidable, let us give an outline why this is true. We will prove that if we could decide this problem then we could decide whether a given Turing machine halts on the empty input. The letters in our finite alphabet will be the nonblank symbols that can appear on the tape of the Turing machine, plus a symbol for each state of the machine. A string in this alphabet containing exactly one letter corresponding to a state of the machine can be viewed as coding the Turing machine at one instant in time by the following convention. The nonblank part of the tape is written from left to write and next to the letter corresponding to the square where the head is, we write the letter corresponding to the state the machine is in. For instance suppose the Turing machine has symbols 0 and 1 and 4 states. We choose a, b, c and d to code these states. If, at an instant in time, the content of the tape is $0110000BBBBBBBBBBBB\ldots$ and the head is in square 3 and is in state 3, we could code this as: $011c000$. Now it is easy to make rewriting rules corresponding to the moves of the machine. For instance if the machine would write 0, go into state 2 and move left when it is in state 3 and sees a 1 this would correspond to the rewriting rule $1c \rightarrow b0$. Now the question whether a machine halts on the empty input corresponds to the question whether we can rewrite a to a description of a halted Turing machine. To make this description unique we add a special state to the Turing machine such that instead of just halting, it erases the tape and returns to the beginning of the tape and then halts. In this case we get a unique halting configuration, which is used as the target string.

It is very interesting to note that although one would expect that the complexity of this problem comes from the fact that we do not know which rewriting rule to apply when there is a choice, this is not used in the proof. In fact in the special cases we get from the reduction from Turing machines, at each point there is only one rule to apply (corresponding to the move of the Turing machine).

In the example given in the exercise there is no way to transform the start string to the target string. This might be seen by letting a have weight 2 and b have weight 1. Then the rewriting rules preserve weight while the two given words are of different weight.

II.10 Undecidable. Do the same reduction as in exercise II.9 to get a rewriting system and a start string corresponding to a Turing machine M_x working on empty input. If this system produces arbitrarily long words then the machine does not halt. On the other hand if we knew that the system did not produce arbitrarily long words then we could simulate the machine until it either halts or enters the same state twice (we know one of these two cases will happen). In the first case the machine halted and in the second it will loop forever. Thus if we could decide if a rewriting system produced arbitrarily long strings we can decide if a Turing machine halts on empty input.

II.11 This problem is decidable. Make a directed graph G whose nodes correspond to the letters in the alphabet. There is an edge from v to w if there is a rewriting rule which rewrites v into a string that contains w. Let the weight of this string be 1 if the rewriting rule replaces v by a longer string and 0 otherwise. Now we claim that the rewriting rules can produce arbitrarily long strings iff there is a circuit of positive weight that can be reached from one of the letters contained in the starting word. The decidability now follows from standard graph algorithms.

3 Efficient computation, hierarchy theorems.

To decide what is mechanically computable is of course interesting, but what we really care about is what we can compute in practice, i.e by using an ordinary computer for a reasonable amount of time. For the remainder of these notes all functions that we will be considering will be recursive and we will concentrate on what resources are needed to compute the function. The two first such resources we will be interested in are computing time and space.

3.1 Basic Definitions

Let us start by defining what we mean by the running time and space usage of a Turing machine. The running time is a function of the input and experience has showed that it is convenient to treat inputs of the same length together.

Definition 3.1 *A Turing machine M runs in time $T(n)$ if for every input of length n, M halts within $T(n)$ steps.*

Definition 3.2 *The length of string x is denoted by $|x|$.*

The natural definition for space would be to say that a Turing machine uses space $S(n)$ if its head visits at most $S(n)$ squares on any input of length n. This definition is not quite suitable under all circumstances. In particular, the definition would imply that if the Turing machine looks at the entire input then $S(n) \geq n$. We will, however, also be interested in machines which use less than linear space and to make sense of this we have to modify the model slightly. We will assume that there is a special input-tape which is read-only and a special output-tape which is write-only. Apart from these two tapes the machine has one or more work-tapes which it can use in the oldfashioned way. We will then only count the number of squares visited on the work-tapes.

Definition 3.3 *Assume that a Turing machine M has a read-only input-tape, a write-only output-tape and one or more work-tapes. Then we will say that M uses space $S(n)$ if for every input of length n, M visits at most $S(n)$ tape squares on its work-tapes before it halts.*

When we are discussing running times we will most of the time not be worried about constants i.e. we will not really care if a machine runs in time n^2 or $10n^2$. Thus the following definition is useful:

Definition 3.4 *$O(f(n))$ is the set of functions which is bounded by $cf(n)$ for some positive constant c.*

Having done the definitions we can go on to see whether more time (space) actually enables us to compute more functions.

3.2 Hierarchy theorems

Before we start studying the hierarchy theorems (i.e. theorems of the type "more time helps") let us just prove that there are arbitrarily complex functions.

Theorem 3.5 *For any recursive function $f(n)$ there is a function V_f which is recursive but cannot be computed in time $f(n)$.*

Proof: Define V_f by letting $V_f(x)$ be 1 if M_x is a legal Turing machine which halts with output 0 within $f(|x|)$ steps on input x and let $V_f(x)$ take the value 0 otherwise.

We claim that V_f cannot be computed within time $f(n)$ on any Turing machine. Suppose for contradiction that M_y computes V_f and halts within time $f(|x|)$ for every input x. Consider what happens on input y. Since we have assumed that M_y halts within time $f(|y|)$ we see that $V_f(y) = 1$ iff M_y gives output 0, and thus we have reached a contradiction.

To finish the proof of the theorem we need to check that V_f is recursive, but this is fairly straightforward. We need to do two things on input x.

1. Compute $f(|x|)$.
2. Check if M_x is a legal Turing machine and in such a case simulate M_x for $f(|x|)$ steps and check whether the output is 0.

The first of these two operations is recursive by assumption while the second can be done using the universal Turing machine as a subroutine. This completes the proof of Theorem 3.5 ∎

Up to this point we have not assumed anything about the alphabet of our Turing machines. Implicitly we have thought of it as $\{0, 1, B\}$ but let us now highlight the role of the alphabet in two theorems.

Theorem 3.6 *If a Turing machine M computes a $\{0, 1\}$ valued function f in time $T(n)$ then there is a Turing machine M' which computes f in time $2n + \frac{T(n)}{2}$.*

Proof: (Outline) Suppose that the alphabet of M is $\{0, 1, B\}$ then the alphabet of M' will be 5-tuples of these symbols. Then we can code every five adjacent squares on the tape of M into a single square of M'. This will enable M' to take several steps of M in one step provided that the head stays within the same block of 5 symbols coded in the same square of M'. However, it is not clear that this will help since it might be the case that many of M's steps will cross a boundary of 5-blocks. One can avoid this by having the 5-tuples of M' be overlapping, and we leave this construction to the reader.

The reason for requiring that f only takes the values 0 and 1 is to make sure that M does not spend most of its time printing the output and the reason for adding $2n$ in the running time of M' is that M' has to read the input in the old format before it can be written down more succinctly and then return to the intitial configuration. ∎

The previous theorem tells us that we can gain any constant factor in running time provided we are willing to work with a larger alphabet. The next theorem tells us that this is all we can gain.

Theorem 3.7 *If a Turing machine M computes a $\{0, 1\}$ valued function f on inputs that are binary strings in time $T(n)$, then there is a Turing machine M' which uses the alphabet $\{0, 1, B\}$ which computes f in time $cT(n)$ for some constant c.*

Proof: (Outline) Each symbol of M is now coded as a finite binary string (assume for notational convenience that the length of these strings is 3 for any symbol of M's alphabet). To each square on the tape of M there will be associated 3 tape squares on the tape of M' which will contain the code of the corresponding symbol of M. Each step of M will be a sequence of steps of M' which reads the corresponding squares. We need to introduce some intermediate states to remember the last few symbols read and there are some other details to take care of. However, we leave these details to the reader. ∎

The last two theorems tell us that there is no point in keeping track of constants when analyzing computing times. The same is of course true when analyzing space since the proofs naturally extend. The theorems also say that it is sufficient to work with Turing machines that have the alphabet $\{0, 1, B\}$ as long as we remember that constants have no significance. For definiteness we will state results for Turing machines with 3 tapes.

It will be important to have efficient simulations and we have the following theorem.

Theorem 3.8 *The number of operations for a universal two-tape Turing machine needed to simulate $T(n)$ operations of a Turing machine M is at most $\alpha T(n) \log T(n)$, where α is a constant dependent on M, but independent of n. If the original machine runs in space $S(n) \geq \log n$, the simulation also runs in space $\alpha S(n)$, where α again is a constant dependent on M, but independent of n.*

We skip the complicated proof.

Now consider the function V_f defined in the proof of Theorem 3.5 and let us investigate how much is needed to compute it. Of the two steps of the algorithm, the second step can be analyzed using the above result and thus the unknown part is how long it takes to compute $f(|x|)$. As many times in mathematics we define away this problem.

Definition 3.9 *A function f is time constructible if there is a Turing machine that on input 1^n computes $f(n)$ in time $f(n)$.*

It is easy to see that most natural functions like $n^2, 2^n$ and $n \log n$ are time constructible. More or less just collecting all the pieces of the work already done we have the following theorem.

Theorem 3.10 *If $T_2(n)$ is time constructible, $T_1(n) > n$, and*

$$\lim_{n \to \infty} \frac{T_2(n)}{T_1(n) \log T_1(n)} = \infty$$

then there is a function computable in time $O(T_2(n))$ but not in $T_1(n)$. Both time bounds refer to Turing machines with three tapes.

Proof: The intuition would be to use the function V_{T_1} defined previously. To avoid some technical obstacles we work with a slightly modified function.

When simulating M_x we count the steps of the simulating machine rather than of M_x. I.e. we first compute $T_2(n)$ and then run the simulation for that many steps. We use two of the tapes for the simulation and the third tape to keep a clock. If we get an answer within this simulation we output 1 if the answer was 0 and output 0 otherwise. If we do not get an answer we simply answer 0. This defines a function V'_{T_2} and we need to check that it cannot be computed by any M_y in time T_1.

Remember that there are infinitely many y_i such that M_{y_i} codes M_y (we allowed an end marker in the middle of the description). Now note that the constant α in Theorem 3.8 only depends on the machine M_y to be simulated and thus there is a y_i which codes M_y such that

$$T_2(|y_i|) \geq \alpha T_1(|y_i|) \log T_1(|y_i|).$$

By the standard argument M_y will make an error for this input. ∎

It is clear that we will be able to get the same result for space-complexity even though there is some minor problems to take care of. Let us first prove that there are functions which require arbitrarily large amounts of space.

Theorem 3.11 *If $f(n)$ is a recursive function then there is a recursive function which cannot be computed in space $f(n)$.*

Proof: Define U_f by letting $U_f(x)$ be 1 if M_x is a legal Turing machine which halts with output 0 without visiting more than $f(|x|)$ tape squares on input x and let $U_f(x)$ take the value 0 otherwise.

We claim that U_f cannot be computed in space $f(n)$. Given a Turing machine M_y which never uses more than $f(n)$ space, then as in all previous arguments M_y will output 0 on input y iff $U_f(y) = 1$ and otherwise $U_f(y) = 0$.

To finish the theorem we need to prove that U_f is recursive. This might seem obvious at first since we can just use the universal machine to simulate M_x and all we have to keep track of is whether M_x uses more than the allowed amount of space. This is not quite sufficient since M_x might run forever and never use more than $f(|x|)$ space. We need the following important but not very difficult lemma.

Lemma 3.12 *Let M be a Turing machine which has a work tape alphabet of size c, Q states and k work-tapes and which uses space at most $S(n)$. Then on inputs of length n, M either halts within time $nQS(n)^k c^{kS(n)}$ or it never halts.*

Proof: Let a *configuration* of M be a complete description of the machine at an instant in time. Thus, the configuration consists of the contents of the tapes of M, the positions of all its heads and its state.

Let us calculate the number of different configurations of M given a fixed input of length n. Since it uses at most space $S(n)$ there at most $c^{kS(n)}$ possible contents of it work-tapes and at most $S(n)^k$ possible positions of the heads on the worktapes. The number of possible locations of the head on the input-tape is at most n and there are Q possible states. Thus we have a total of $nQS(n)^k c^{kS(n)}$ possible configurations. If the machine does not halt within this many timesteps the machine will be in the same configuration twice. But since the future actions of the machine is completely determined by the present configuration, whenever it returns to a configuration where it has been previously it will return infinitely many times and thus never halt. The proof of Lemma 3.12 is complete. ∎

Returning to the proof of Theorem 3.11 we can now prove that U_f is computable. We just simulate M_x for at most $|x|Qf(|x|)^k c^{kf(|x|)}$ steps or until it has halted or used more than $f(|x|)$ space. We use a counter to count the number of steps used. This finishes the proof of Theorem 3.11 ∎

To prove that more space actually enables us to compute more functions we need the appropriate definition.

Definition 3.13 *A function f is space constructible if there is a Turing machine that on input 1^n computes $f(n)$ in space $f(n)$.*

We now can state the space-hierarchy theorem.

Theorem 3.14 *If $S_2(n)$ is space constructible, $S(n) \geq \log n$ and*

$$\lim_{n\to\infty} \frac{S_2(n)}{S(n)} = \infty$$

then there is a function computable in space $O(S_2(n))$ but not in space $S(n)$. These space bounds refer to machines with 3 tapes.

Proof: The function achieving the separation is basically U_S with the same twist as in Theorem 3.10. In other words define a function essentially as U_S but restrict the computation to using space S_2 of the simulating machine. The rest of the proof is now more or less identical. The only detail to take care of is that if $S(n) \geq \log n$ then a counter counting up to $|x|QS(|x|)^k c^{kS(|x|)}$ can be implemented in space $S(n)$. ∎

The reason that we get a tighter separation between space-complexity classes than time-complexity classes is the fact that the universal machine just uses constant more space than the original machine.

This completes our treatment of the hierarchy theorems. These results are due to Hartmanis and Stearns and are from the 1960's. Next we will continue into the 1970's and move further away from recursion theory and into the realm of more modern complexity theory.

4 The complexity classes L, P and $PSPACE$.

We can now start our main topic, namely the study of complexity classes. We will in this section define the basic deterministic complexity classes, L, P and $PSPACE$.

Definition 4.1 *Given a set A, we say that $A \in L$ iff there is a Turing machine which computes the characteristic function of A in space $O(\log n)$.*

Definition 4.2 *Given a set A, we say that $A \in P$ iff there is a Turing machine which for some constant k computes the characteristic function of A in time $O(n^k)$.*

Definition 4.3 *Given a set A, we say that $A \in PSPACE$ iff there is a Turing machine which for some constant k computes the characteristic function of A in space $O(n^k)$.*

There are some relations between the given complexity classes.

Theorem 4.4 *$L \subset PSPACE$.*

Proof: The inclusion is obvious. That it is strict follows from Theorem 3.14. ∎

Theorem 4.5 *$P \subseteq PSPACE$.*

This is also obvious since a Turing machine cannot use more space than time.

Theorem 4.6 *$L \subseteq P$.*

Proof: This follows from Lemma 3.12 since if $S(n) \leq c \log n$ and we assume that the machine uses a three letter alphabet, has k work-tapes, and Q states and always halts, then we know it runs in time at most

$$nQ(c \log n)^k 3^{c \log n} \in O(n^{2+c \log 3})$$

where we used that $(\log n)^k \in O(n)$ for any constant k. We can conclude that a machine which runs in logarithmic space also runs in polynomial time. ∎

The inclusions given in Theorems 4.5 and 4.6 are believed to be strict but this is not known. Of course, it follows from Theorem 4.4 that at least one of the inclusions is strict, but it gives no idea to which one it is.

Figure 2: A Random Access Machine

4.1 Is the definition of P model dependent?

When studying mechanically computable functions we had several definitions which turned out to be equivalent. This fact convinced us that we had found the right notion i.e. that we had defined a class of functions which captured a property of the functions rather than a property of the model. The same argument applies here. We have to investigate whether the defined complexity classes are artifacts of the particulars of Turing machines as a computational model or if they are genuine classes of functions which are more or less independent of the model of computation. The reader who is not worried about such questions is adviced to skip this section.

Turing machine seems incredibly inefficient and thus we will compare it to a model of computation which is more or less a normal computer (programmed in assembly language). This type of computer is called a Random Access Machine (RAM) and a pictured is given i Figure 2. A RAM has a finite control, and infinite number of registers and two accumulators. Both the registers and the accumulators can hold arbitrarily large integers. We will let $r(i)$ be the content of register i and ac_1 and ac_2 the contents of the accumulators. The finite control can read a program and has a read-only input-tape and a write-only output tape. In one step a RAM can carry out the following instructions.

1. Add, subtract, divide (integer division) or multiply the two numbers in ac_1 and ac_2, the result ends up in ac_1.
2. Make conditional and unconditional jumps. (Condition $ac_1 > 0$ or $ac_1 = 0$).
3. Load something into an accumulator, e.g. $ac_1 = r(k)$ for constant k or $ac_1 = r(ac_1)$, similarly for ac_2.
4. Store the content of an accumulator, e.g. $r(k) = ac_1$ for constant k or $r(ac_2) = ac_1$, similarly for ac_2.
5. Read input $ac_1 = input(ac_2)$.
6. Write an output.
7. Use constants in the program.
8. Halt

One might be tempted to let the time used by a RAM be the number of operations it does (the *unit-cost RAM*). This turns out to give a quite unrealistic measure of complexity and instead we will use the logarithmic cost model.

Definition 4.7 *The time to do a particular instruction on a RAM is* $1 + \log(k + 1)$ *where* k *is the least upper bound on the integers involved in the instruction. The time for a computation on a RAM is the sum of the times for the individual instructions.*

This actually agrees quite well with our everyday computers. The size of a computer word is bounded by a constant and operations on larger numbers require us to access a number of memory cells which is proportional to logarithm of the number used.

To define the amount of memory used by a RAM on a particular operation let us assume that the initial contents of all the registers are 0. Then we have:

Definition 4.8 *The space used by a RAM under a computation is the maximum of*

$$\log(ac_1 + 1) + \log(ac_2 + 1) + \sum_{r(i) \neq 0} \log\left(i + r(i)\right)$$

during the computation.

Intuitively the RAM seems more powerful than a Turing machine. We will not try to prove exactly this, but only to establish strong enough results to show that the class P is well defined.

Theorem 4.9 *If a Turing machine can compute a function in time $T(n)$ and space $S(n)$, for $T(n) \geq n$ and $S(n) \geq \log n$ then the same function can be computed in time $O(T^2(n))$ and space $O(S(n))$ on a RAM.*

Proof: (Outline) Assume for simplicity that the Turing machine just has one work-tape and that it uses the alphabet $\{0, 1, B\}$. The RAM will simulate the computation of the Turing machine step by step. It will code the content of the work-tape as an integer and store this integer in register 1, the position of the head on the input-tape in accumulator 2, the position of the head on the work-tape(s) in register 2 and the current state of the Turing machine in register 3. To simulate a step of the Turing machine the RAM gets the appropriate information from the work-tape by an integer division and then it follows the transition described by the next-step function. The cost of the simulation of an individual step is the size of the integers involved and this is bounded by $O(S(n))$. Since we have at most $T(n)$ steps and $S(n) \leq T(n)$ the bound for the running time follows. The bound for the space used is obvious. ∎

Observe that we need to store the entire contents of the work-tape in one register to conserve space. If we instead stored the content of square i in register i the total space used would be $O(S(n) \log S(n))$. The running time would be improved to $O(T(n) \log T(n))$ but for the present purposes it is more important to keep the space small.

Next let us see that in fact a Turing machine is not that much less powerful than a RAM.

Theorem 4.10 *If a function f can be computed by a RAM in time $T(n)$ and space $S(n)$ then f can be computed in time $O(T^2(n))$ and space $S(n)$ on a Turing machine.*

Proof: (Outline) As many other proofs this is not a very thrilling simulation argument, which we usually tend to omit. However, since the result is central in that it proves that P is invariant under change of model, we will at least give a reasonable outline of the proof.

The way to proceed is of course to simulate the RAM step by step. Assume for simplicity that we do the simulation on a Turing machine which apart from its input-tape and output-tape has 4 work-tapes. Three of the four work-tapes will correspond to ac_1, ac_2, the registers, respectively, while the forth tape is used as a scratch pad. A schematic picture is given in Figure 3 The register tape will contain pairs $(i, r(i))$ where the two numbers are separated with a B. Two different pairs are separated by BB. If some i does not appear on the register tape this means that $r(i) = 0$.

The RAM-program is now translated into a next-step function of the Turing machine. Each line is translated into a set of states and transitions between the states as indicated by Figure 4. Let us give a few examples how to simulate some particular instructions. We will define the Turing machine pictorially by having circles indicate states. Inside the circle we write the tape(s) we are currently interested in, and the labeled arrows going out of the circle indicate which states to proceed to where the label indicates the current symbol(s). Rectangular boxes indicate subroutines, a special subroutine is "Rew" which is rewinding the register tape, i.e. moving the head to the beginning, the same operation also applies to other tapes.

1. If the instruction is an arithmetical step, we just replace it by a Turing machine which computes the arithmetical step using the ac_1 and ac_2 tapes as inputs and the scratch pad tape as work-tape.

2. If the instruction is jump-instruction we just make the next-step function take the next state which is the first state of the set of states corresponding to that line. (See Figure 5.)

3. If the jump is conditional on the content of ac_1 being 0, then we just search the ac_1-tape for the symbol 1. If we do not find any 1 before we find B the next step function directs us to the given line and otherwise we proceed with the next line. (See Figure 6.)

Figure 3: A TM simulating a RAM

Figure 4: Basic picture

Figure 5: The jump instruction

Figure 6: Conditional jump

4. Let us just give an outline of how to load $r(ac_2)$ into ac_1. Clearly, what we want to do is to look for the content of ac_2 as the first part of any pair on the register tape. If we find that no such pair exists then we should load 0 into ac_1. A description of this is given in Figure 7.

5. Finally let us indicate how to store ac_1 into register ac_2. To do this we scan the register-tape to find out the present value of $r(ac_2)$. If $r(ac_2) = 0$ previously this is easy. If $ac_1 \neq 0$ we store the pair (ac_2, ac_1) at the end of the register-tape and otherwise we do nothing. If $r(ac_2) \neq 0$ we erase the old copy $(ac_2, r(ac_2))$ and then move the rest of the content of the register-tape left to avoid empty space. After we have moved the information we write (ac_2, ac_1) at the end (provided $ac_1 \neq 0$).

Let us analyze the efficiency of the simulation. The space used by the Turing machine is easily seen to be bounded by

$$O(\log(ac_1 + 1) + \log(ac_2 + 1) + \sum_{r(i) \neq 0} (\log(i + 1) + \log(r(i) + 1) + 3))$$

and thus the simulation works in $O(S(n))$ space. To analyze the time needed for the simulation we claim that you can do multiplication and integer division of two m-digit numbers in time $O(m^2)$ on a Turing machine. This implies that any arithmetical operation can be done in a factor $O(S(n))$ longer

Figure 7: Loading instruction

time on the Turing machine than on the RAM. The storing and retrieving of information can also be done in time $O(S(n))$ and using $S(n) \leq T(n)$ Theorem 4.10 follows. ∎

Using Theorems 4.9 and 4.10 we see that P, L and $PSPACE$ are the same whether we use Turing machines or RAMs in the definitions. This turns out to be true in general and this gives us a very important principle which we can formalize as a complexity theoretic version of Church's thesis.

COMPLEXITY THEORETIC VERSION OF CHURCH'S THESIS: *The complexity classes* L, P *and* PSPACE *remain the same under any reasonable computational model.*

The above statement also remains true for all other complexity classes that we will define throughout these notes and we will frequently implicitly apply the above thesis. This works as follows. When designing algorithms it is much easier to describe and analyze the algorithm if we use a high level description. On the other hand when we argue about computation it is much easier to work with Turing machines since their local behavior is so easy to describe. By virtue of the above thesis we can take the easy road to both things and still be correct.

4.2 Examples of members in the complexity classes.

We have defined L, P and $PSPACE$ as families of sets. We will every now and then abuse this notation and say that a function (not necessarily $\{0,1\}$-valued) lies in one of these complexity classes. This will just mean that the function can be computed within the implied resource bounds.

Example 4.11 Given two n-digit numbers x and y written in binary, compute their sum.

This can clearly be done in time $O(n)$ as we all learned in first grade. It is also quite easy to see that it can be done in logarithmic space. If we have $x = \sum_{i=0}^{n-1} x_i 2^i$ and $y = \sum_{i=0}^{n-1} y_i 2^i$ then $x + y$ is computed by the following program:

$carry = 0$
For $i = 0$ to $n - 1$
    $bit = x_i + y_i + carry$
    $carry = 0$
    If $bit \geq 2$ then $carry = 1, \ bit = bit - 2.$

write bit
next i
write $carry$.

The only things that need to be remembered is the counter i and the values of bit and $carry$. This can clearly be done in $O(\log n)$ space and thus addition belongs to L.

Example 4.12 Given two n-digit numbers x and y written in binary, compute their product.

This can again be done in P by first grade methods, and if we do it as taught, it will take us $O(n^2)$ (this can be improved by more elaborate methods). In fact we can also do it in L.

$carry$= 0
For $i = 0$ to $2n - 2$
$low = \max(0, i - (n-1))$
$high = \min(n-1, i)$
For $j = low$ to $high$, $carry = carry + x_j * y_{i-j}$
write $lsb(carry)$
$carry = carry/2$
next i
write $carry$ with least significant bit first

If one looks more closely at the algorithm one discovers that it is the ordinary multiplication algorithm where one saves space by computing a number only when it is needed. The only slightly nontrivial thing to check in order to verify that the algorithm does not use more that $O(\log n)$ space is to verify that $carry$ always stays less than $2n$. We leave this easy detail to the reader.

One might be tempted to think that also division could be done in L. However, it is not known whether this is the case. Another very easy problem that is not known how to do in L: Given an integer in base 2, convert it to base 3.

Example 4.13 Given two n-bit integers x and y compute their greatest common divisor.

We will show that this problem is in P and in fact give two different algorithms to show this. First the old and basic algorithm: Euclid's algorithm. Assume for simplicity that $x > y$.

$a = x$
$b = y$
While $b \neq 0$ do
  find q and r such that $a = bq + r$, $0 \leq r < b$
  $a = b$
  $b = r$
od
write a

The algorithm is correct since if d divides x and y then clearly d divides all a and b. On the other hand if d divides any pair a and b then it also divides x and y.

To analyze the algorithm we have to focus on two things, namely the number of iterations and the cost of each iteration. First observe that for each iteration the numbers get smaller and thus we will always be working with numbers with at most n digits. The work in each iteration is essentially a division and this can be done in $O(n^2)$ bit operations.

The fact that numbers get smaller at each iteration implies that there are at most 2^n iterations. This is not sufficient to get a polynomial time running time and we need the following lemma.

Lemma 4.14 *Let a and b have the values a_0 and b_0 at one point in time in Euclid's algorithm and let a_2 and b_2 be their values two iterations later, then $a_2 \leq a_0/2$.*

Proof: Let a_1 and b_1 be the values of a and b after one iteration. Then if $b_0 < a_0/2$ we have $a_2 < a_1 = b_0 < a_0/2$ and the conclusion of the lemma is true. On the other hand if $b_0 \geq a_0/2$ then we will have $a_2 = b_1 = a_0 - b_0 \leq a_0/2$ and thus we have proved the lemma. ∎

The lemma implies that there are at most $2n$ iterations and thus the total complexity is bounded by $O(n^3)$. If you are careful, however, it is possible to do better (without applying any fancy techniques) by the following observation. If you use standard long division (with remainder) to find q then the complexity is actually $O(ns)$ where s is the number of bits in q.

Thus if q is small we can do each iteration significantly faster. On the other hand if q is large then it is easy to see that the numbers decrease more rapidly than given by the above lemma. If one analyzes this carefully we actually get complexity $O(n^2)$.

Let us give another algorithm for the same problem. This algorithm is called "Binary GCD".

Let 2^{d_x} be the highest power of 2 that divides x and define d_y similarly.
Set $a = x2^{-d_x}$ and $b = y2^{-d_y}$. If $b < a$ interchange a and b.
While $b > 1$ do
    Either $a + b$ or $a - b$ is divisible by 4. Set r to the number that is divisible by 4 and set $a = \max(b, r2^{-d_r})$ and $b = \min(b, r2^{-d_r})$ where 2^{d_r} is the highest power of 2 that divides r.
od
write $a2^{\min(d_x,d_y)}$

The algorithm is correct by a similar argument as the previous algorithm. To analyze the complexity of the algorithm we again have to study the number of iterations and the cost of each iteration. Again it is clear that the numbers decrease in size and thus we will never work with numbers with more than n digits. Each iteration only consists of a few comparisons and shifts if the numbers are coded in binary and thus it can be implemented in time $O(n)$. To analyze the number of iterations we have:

Lemma 4.15 *Let a and b have the values a_0 and b_0 at one point in time in the binary GCD algorithm and let a_2 and b_2 be their values two iterations later, then $a_2 \leq a_0/2$.*

Proof: If a_1 and b_1 are the numbers after one iteration then $b_1 \leq (a_0+b_0)/4$ and $a_1 \leq a_0$. Since $b_0 \leq a_0$ this implies that $a_2 \leq \max(b_1, (a_1 + b_1)/4) \leq a_0/2$. ∎

Thu again we can conclude that we have at most $2n$ iterations, and hence the total work is bounded by $O(n^2)$. This implies that binary GCD is a competitive algorithm in particular since the individual operations can be implemented very efficiently when the binary representation of integers is used.

Let us just remark that the best known greatest common divisor algorithm for integers runs in time $O(n(\log n)^2 \log\log n)$ and is based on the

Euclidean algorithm. It is unknown if integer greatest common divisor can be solved in small space.

Example 4.16 Given a nonsingular integer matrix M with entries which are n-bit numbers, solve $Mx = b$ for some vector of n-bit numbers.

It might seem like this problem obviously is in P since Gaussian elimination is well known to be doable in $O(n^3)$ steps. However, there is something to check. We need to verify that the numbers do not get too large during the computation i.e. that the rational numbers that appear can be represented. To analyze what happens to the numbers assume for notational simplicity that the upper left $i \times i$ matrix is non-singular for any i and thus we will be able to perform Gaussian elimination without pivoting. Let us investigate what the matrix looks like after we have eliminated the i'th variable. Suppose the original matrix looks like

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

where A is the upper $i \times i$ matrix. After the i'th variable has been eliminated the matrix will be

$$\begin{pmatrix} A^{-1} & 0 \\ -CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & A^{-1}B \\ 0 & -CA^{-1}B + D \end{pmatrix}$$

where I is the $i \times i$ identity matrix. Thus using the following lemma we can bound the rational numbers involved in the computation.

Lemma 4.17 *If A is a nonsingular $n \times n$ integer matrix with entries bounded in size by m then A^{-1} has rational entries with numerator and denominator bounded by $m^n n^{\frac{n}{2}}$.*

Proof: Any entry of A^{-1} is an $(n-1) \times (n-1)$ subdeterminant of A divided by the determinant of A. Thus we just need to bound the size of determinants of integer matrices. A determinant can be interpreted as the volume of the parallelipiped spanned by the rows. This volume is bounded by the product of the lengths of the row vectors[3] which in its turn is bounded by $(m\sqrt{n})^n$. ■

[3] This is not a formal proof and the inequality indicated in this sentence is known as Hadamard's inequality

It follows from the lemma that the rational numbers involved in Gaussian elimination can be represented by $O(n^2)$ binary digits. Since Gaussian elimination can be done in $O(n^3)$ operations and each operation can be performed in time $O(n^4)$ (if we use classical arithmetic) then we get total complexity $O(n^7)$.

Example 4.18 The determinant of an $n \times n$ matrix can be written as

$$\sum_{\pi \in S_n} sg(\pi) \prod_{i=1}^{n} x_{i,\pi(i)}$$

where the sum is over all permutations of the numbers 1 through n and $sg(\pi)$ is the signum[4] of the permutation. The determinant can be computed by Gaussian elimination and thus by the previous example it is in P. The *permanent* is a very related number which is defined as

$$\sum_{\pi \in S_n} \prod_{i=1}^{n} x_{i,\pi(i)}.$$

Thus we have just removed the signum part of the definition. The definition looks simpler but it removes the nice invariance under the row operations of Gaussain elimination. There is no known polynomial time algorithm for computing the permanent and there is good reason to believe that there is no such algorithm (the problem is $\#P$-complete, we will get to this complexity class later). It is not hard to see that the problem is in *PSPACE* and we will not give the most efficient algorithm but rather the easiest to understand.

$per = 0$
For $1 \leq \pi(1), \pi(2), \ldots \pi(n) \leq n$
  If $\pi(i) \neq \pi(j)$ for $i \neq j$, $per = per + \prod_{i=1}^{n} x_{i,\pi(i)}$

Thus we just generate all n-tuples of numbers between 1 and n, check if it is a permutation and, if it is, add the corresponding term to the sum. All the space required is to store the variables $\pi(i)$ and per. The space needed for the former is bounded by $(n \log n)$ while the second is bounded by the size of the answer and if we assume that all entries in the original matrix are bounded by 2^n the per is bounded by $2^{n^2} n!$ and thus can be stored in space $O(n^2)$.

[4] If you do not know the signum function just forget this definition of the determinant

It is interesting to note that there is a polynomial time algorithm to decide whether a permanent of a $0,1$ matrix is nonzero but that it seems hard to compute it.

Example 4.19 Given a prime number p and a number a, find x (if one exists) such that $x^2 \equiv a \pmod{p}$.

Let us first recall some basic facts from number theory. Assume that we have an odd prime p (the case $p = 2$ being easy) then a can be written as a square mod p iff $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ (i.e. we have a solution iff this condition holds). Remember also that, by Fermat's little theorem, it is true that $x^{p-1} \equiv 1 \pmod{p}$ for any number not divisible by p.

Now if $p \equiv 3 \pmod{4}$ and if $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ then if we set $x = a^{\frac{p+1}{4}}$ we have

$$x^2 \equiv a^{\frac{p+1}{2}} \equiv aa^{\frac{p-1}{2}} \equiv a. \quad (mod\ p)$$

Thus taking square roots when $p \equiv 3 \pmod{4}$ is just computing a power. Let us investigate how much resources are needed to compute $a^{\frac{p+1}{4}}$ (mod p). Assume that p and a are at most n digit numbers. Then computing $a^{\frac{p+1}{4}}$ by successive multiplications would require on the order of 2^n multiplications. It is more efficient to first compute a^{2^i} (mod p) for $0 \leq i \leq n$ in n squarings. Observe here that since we are only interested in the result (mod p), we can reduce mod p after each squaring and thus we will never need to work with numbers with more than $2n$ digits. Now we write $\frac{p+1}{4}$ in binary and we compute $a^{\frac{p+1}{4}}$ by multiplying together the powers a^{2^i} with the i's corresponding to 1's in the binary expansion of $\frac{p+1}{4}$. Hence, we get $O(n)$ multiplications of $O(n)$ bit numbers and this can be done in total time $O(n^3)$.

Thus we have proved that taking square-roots modulo primes p with $p \equiv 3 \pmod{4}$ can be done in polynomial time. It is not known if this is true in general for primes $p \equiv 1 \pmod{4}$ or when p is a composite number. We will return to these questions later in these notes.

Example 4.20 Given a directed graph G with n nodes and two distinguished nodes s and t in G. Is it possible to find a directed path from s to t?

This problem is in P by the following straightforward algorithm.

Set $R = \{s\}$.
Set R_{new} to the set of nodes reachable from s in one step,
i.e. the set of v such that there is an edge (s, v).

While R_{new} is not empty do

Take an element w in R_{new} and move it into R. Also take any nodes reachable from w in one step which do not belong to either R or R_{new} and put them into R_{new}.

od

If $t \in R$ say yes, otherwise say no.

We claim that when the algorithm ends all the nodes reachable from s are in R. We leave the verification of this to the reader. It is important to know that R_{new} contains the set of nodes known to be reachable from s but whose neighbors have not yet been put into R or R_{new}.

Remark 4.21 *Observe that in fact R is the set of nodes reachable from s and thus we have really solved a more general problem.*

To see that the problem is in P let us analyze the time needed for the algorithm. Since we each time the loop is executed put one node into R and we never remove anything from R the loop is only executed n times. Each execution in the loop can be done in time n since we just have to investigate the neighbors of w. Thus the complexity is bounded by $O(n^2)$.

Next we turn to the definition of non-deterministic computation. The obvious goal in mind is to formally define NP.

5 Nondeterministic computation

The two most famous complexity classes are probably P and NP. We have already defined P and to define NP we need the concept of a nondeterministic Turing machine. The formal definition might make nondeterminism seem like a paper-tiger which has nothing to do with reality, but it will soon be clear that this is not the case.

5.1 Nondeterministic Turing machines

The heart of a normal, deterministic Turing machine is the next-step function, which tells the machine what to do in a given situation. A nondeterministic Turing machine also has a next-step function, but it is multivalued. By this we mean that in a given situation the machine might do several different things. This implies that on a given input there are several possible computations and in particular, there might be several different possible outputs. This calls for a definition.

Definition 5.1 *A nondeterministic Turing machine can only compute functions which takes the values* 0 *and* 1*. The machine takes the value 1 on (or accepts) an input x iff there is some possible computation on input x which gives output 1. If there is no computation that gives the output 1, the machine takes value 0 (or rejects the input).*

Since we will only be working with $\{0, 1\}$ functions we will think of nondeterministic machines as recognizing sets i.e. the set of inputs for which there is an accepting computation.

Example 5.2 Suppose we want to recognize composite numbers i.e. numbers which are not prime and hence can be written as the product of two numbers both greater than or equal to 2. This can be done by a nondeterministic machine as follows:

On input x, write y_1 and y_2 nondeterministically with $|y_i| \leq |x|$ for $i = 1, 2$. Writing down y_1 is done by allowing the machine move left for $|x|$ steps while at each step either writing down 0,1 or an endmarker. The machine constructs y_2 in the same way. Now the machine gives output 1 iff $y_1 y_2 = x$ and $y_i > 1$ for $i = 1, 2$.

Let us see that the algorithm is correct. If x is composite then there is some computation that outputs 1, namely if $x = ab$ then when $y_1 = a$ and $y_2 = b$ we will get the output 1. On the other hand if x is prime there is no possible computation that gives output 1 since if $y_1 y_2 = x$ then by the definition of prime one of the y_i is 1.

Observe that when we are considering deterministic computation recognizing primes and recognizing composite numbers are very similar, since one just changes the output routine to reverse the meaning of 0 and 1. When it comes to nondeterministic computation there is a tremendous difference. If, for instance, you change the output of the machine recognizing composite numbers defined above then you get a machine that accepts everything. It is important to keep this non-symmetry in mind.

The definitions of space and time need to be slightly modified since there is no unique computation given the input.

Definition 5.3 *A nondeterministic Turing machine M runs in time $T(n)$ if for every input of length n, every computation of M halts within $T(n)$ steps.*

Definition 5.4 *A nondeterministic Turing machine M runs in space $S(n)$ if for every input of length n, every computation of M visits at most $S(n)$ squares on the work-tape.*

Since non-deterministic Turing machines can always be made to have output 1 or 0 the size of the answer will always be small. This implies that we do not need an output-tape. Some proofs will be formally easier if we assume that the output is written on the worktape and therefore we will assume this.

With these basic definitions done we can proceed to define some complexity classes.

Definition 5.5 *Given a set A, we say that $A \in NL$ iff there is a nondeterministic Turing machine which accepts A and runs in space $O(\log n)$.*

Definition 5.6 *Given a set A, we say that $A \in NP$ iff there is a nondeterministic Turing machine which accepts A and runs in time $O(n^k)$ for some constant k.*

Definition 5.7 *Given a set A, we say that $A \in NPSPACE$ iff there is a nondeterministic Turing machine which accepts A and runs in space $O(n^k)$ for some constant k.*

We have similar theorems to 4.4, 4.5 and 4.6.

Theorem 5.8 *$NL \subset NPSPACE$.*

Proof: The inclusion is obvious. It is at this point not clear that it is strict. This will follow from results later on and we leave it for the time being. ∎

Theorem 5.9 *$NP \subseteq NPSPACE$.*

Proof: This follows since also nondeterministic Turing machines cannot use more space than time. ∎

Theorem 5.10 *$NL \subseteq NP$.*

Proof: The proof is quite close to the proof of the corresponding deterministic statement but we need an extra observation. The time bound given in Lemma 3.12 is no longer true for nondeterministic computation. The reason for this is that even if a nondeterministic machine is in the same configuration twice it need not loop forever. The reason is that it can make different non-deterministic choices the second time around. However, it is easy to see that if a nondeterministic machine has an accepting computation then it has a nondeterministic computation which visits each configuration at most once. This implies that we can impose the time-restriction given by Lemma 3.12 without changing the set of inputs accepted. This proves Theorem 5.10. ∎

Let us now proceed to some examples of members in the newly defined complexity classes.

Example 5.11 Composite numbers are in *NP*, since the nondeterministic algorithm given previously is easily seen to run in time $O(n^2)$.

It might be tempting to guess that Composite numbers are in *NL* since the essential part of the algorithm is a multiplication and we know from before that multiplication can be done in L. This is not known however, and the reason that the given algorithm does not work is that multiplication is in L only when the input is on a separate input-tape where we can access any part of the input when it is needed. In the present situation we have to write down the two factors on the work-tape and there is no room to do this.

Example 5.12 Traveling Sales Person (TSP): Given n cities and a symmetric integer $n \times n$ matrix $(m_{ij})_{i,j=1}^n$ where m_{ij} denotes the distance between cities i and j, and an integer K. Is there a tour which visits all cities exactly once and is of total length $\leq K$? TSP is in *NP* as can be seen from the following non-deterministic algorithm.

1. Nondeterministically write numbers b_i, $i = 1, 2 \ldots, n$ each with at most $\log n + 1$ digits.
2. If $1 \leq b_i \leq n$ for all i and $b_i \neq b_j$ for $i \neq j$ then compute $\sum_{i=1}^{n-1} m_{b_i, b_{i+1}} + m_{b_n, b_1}$. If this number is less than K output 1 and in all other cases output 0.

Observe that the conditions $1 \leq b_i \leq n$ and $b_i \neq b_j$ for $i \neq j$ imply that the b_i define a tour starting in b_1 and tracing through b_i for increasing i and then returning to b_1. If this tour is short enough the machine accepts the output. It is easy to check that the algorithm runs in polynomial time and thus we have proved that $TSP \in NP$.

Example 5.13 Boolean formula satisfiability: Given a Boolean formula, consisting of Boolean variables x_i, $i = 1, 2 \ldots n$, $\wedge$-gates (logical conjunction) $\vee$-gates (logical disjunction) and negation-gates, is there a setting of the variables that satisfies the formula?

This problem is in *NP*, by the obvious procedure. Namely, nondeterministically write down the value of every variable and then write 1 iff the guessed assignment satisfies the formula. To check that this procedure runs in polynomial time one has to observe that given a formula and an assignment of all the variables then one can check whether the assignment satisfies the formula in polynomial time. This is easy and we leave this as an exercise.

Let us return to the problem of graph-reachability (previously considered in section 4.2:

Example 5.14 Directed graph reachability: Given a directed graph G and two nodes s and t of G, is there a directed path from s to t?

We present an algorithm that uses only logarithmic space and hence we need to be slightly careful about how the input is presented. We assume that the graph is given as a list of the edges. Now we have the following algorithm:

Suppose the graph has n nodes.

Set $H = s$
For $i = 1, 2 \ldots n$

    If $H = t$ print 1 and halt.
    If there is no edge out of H print 0 and halt.
    Choose nondeterministically one of the edges leaving H and set H to the endpoint of this edge.

Next i
Print 0.

This procedure uses only logarithmic space since all we need to remember is the counter i and the value of H. The conditions given in the algorithm are easily checked given the assumed encoding of G.

To verify that the algorithm is correct, first observe that by construction H is always a node that can be reached from s. Thus, since the machine output 1 only when $H = t$, we know that when the machine takes the value 1 then t is reachable from s. On the other hand suppose that t is reachable from s. Then there is a path $v_1, v_2 \ldots v_k$ where $v_1 = s$, $v_k = t$ and there is an edge from v_i to v_{i+1} for any i. We can assume that $k \leq n$ since if $v_i = v_j$ for $i < j$ then we can eliminate v_{i+1} through v_j and still maintain a path. Then there is a possibility that $H = v_i$ for every i and thus there is a possibility that the machine outputs 1.

The argument implies that the algorithm recognizes exactly the graphs that have a path from s to t and therefore directed graph reachability is in *NL*.

We will not give any example of a language in *NPSPACE* and in the next section it will be clear why.

Before we continue to establish some of the more formal properties of *NP*, let us be informal for a while.

The class P is intuitively thought of as the class of functions which are computable in practice, i.e. within moderate amounts of computation we can solve reasonably large problems. That this is the case is not clear from the definition and one could object that although n^{100} is polynomial, it grows too quickly. In practice however, this anomaly does not seem to appear and thus if a problem has a polynomial time solution then the exponent tends to be small and the algorithm is usually efficient in practice.

In a similar way *NP* can be thought of as the class of problems where, if you knew the solution, it could be verified efficiently. In an abstract meaning "the solution" must here be interpreted as the set of nondeterministic choices that makes the machine accept. As we have seen, in practice "the solution" is much more concrete. Thus the nondeterministic choices have in our examples corresponded to the factors, a short tour, and a satisfying assignment, respectively.

The recursive sets corresponded to functions that could be computed, while the recursively enumerable sets corresponded to statements that could be verified. The latter statement follows from the fact that if A is r.e. and $x \in A$ then this can be verified since we just wait until x is listed. On the other hand if $x \notin A$ this cannot be verified since we never know if we just haven't waited long enough to see it listed. In view of this one can say that recursive and r.e. have the same relation as P and NP and thus it is not surprising that we can prove some similar theorems.

Theorem 5.15 *Given a set A, then $A \in NP$ iff there is a language $B \in P$ and a constant k such that*

$$x \in A \Leftrightarrow \exists_{y,|y| \leq |x|^k} (x, y) \in B.$$

Proof: Let us first prove that if there is such a B then $A \in NP$. In fact, a nondeterministic algorithm for membership in A just consists of guessing a y of the desired length and then accepting iff $(x, y) \in B$. If B can be recognized in time $O(n^c)$ this procedure runs in time $O(n^{(1+k)c})$ which is polynomial.
To see the converse, we will need the concept of a computation tableau.

Definition 5.16 *A* computation tableau *is a complete description of a computation of a Turing machine. It consists of all configurations of the Turing machine on a specific input (i.e one configuration for every time step) starting with the input configuration and ending with the halting configuration.*

The reason for the name is that we will think of it in the following way. Assume that the Turing machine has only one tape. Then we can think of its computation tableau as a two-dimensional array with time on one axis and the tape squares on the other. The position (i, j) of this tableau thus contains the symbol that is in the j'th square at time i. It also contains information about whether the head is there and in such a case which state it is in. A computation which starts with input $x_1,\ x_2 \ldots x_n$ on the input-tape and ends with only a 1 on the tape is given in Table 3.

x_1, q_0	x_2	x_3	x_4	...
0	x_2, q_3	x_3	x_4	...
0	1	x_3, q_1	x_4	...
.		.		
.		.		
$1, q_h$	B	B	B	B

Table 3: A computation tableau

Now, we can return to the converse of Theorem 5.15. Suppose A is recognized by a one-tape Turing machine M in nondeterministic time n^c. Define B to be the set of pairs (x, y) such that y describes an $n^c \times n^c$ computation tableau of M on input x which ends in an accepting state. Then B satisfies the condition with respect to A of the theorem with $k = 2c$. We claim that B is in P. To see this observe that to check whether a pair (x, y) is in B we basically have to check three things.

1. That the computation described by y starts with x on the input tape.
2. That the computation is legal for M.
3. That the computation accepts.

The first and the last conditions are easy to check since they just talk about the contents of particular squares. Also to check 2 is straightforward since we have to check that the only square that changed value between two timesteps is the square where the head was located, and also that the transition by the head was a possible transition given the next-step function of M. This finishes the proof. ∎

Remark 5.17 *One might be tempted to think that the relation given between* NP *and* P *in Theorem 5.15 would be true also for* NL *and* L*. As the interested reader can convince himself, this is probably not the case as even if we restrict* B *to belong to* L *then the set of all* A *definable in this way is still all of* NP*.*

Thus we have given the theorem about NP and P corresponding to Theorem 2.19. Of the other theorems in Section 2.7, it is not known whether the analogue of 2.17 is true. (The general belief is that it is not.) There is a nice reduction theory and also a notion of complete sets and we will return to these questions in Chapter 7.

6 Relations among complexity classes

Up to this point we have defined six complexity classes *(L,P,PSPACE, NL,NP,* and *NPSPACE)* and we have observed some relations. In this section we will establish some more relations, some obvious and some not obvious. Let us first observe that the option of non-determinism will never hurt and thus any deterministic complexity class is contained in the corresponding nondeterministic complexity class. This gives us three immediate theorems.

Theorem 6.1 $L \subseteq NL$.

Theorem 6.2 $P \subseteq NP$.

Theorem 6.3 $PSPACE \subseteq NPSPACE$.

In the next subsection we will prove the first nontrivial complexity result. For notational convenience let $TIME(T(n))$ denote the class of languages that can be recognized in deterministic time $T(n)$ and let $NTIME(T(n))$ be the class of languages that can be recognized in the same nondeterministic time. Similarly we define $SPACE(S(n))$ and $NSPACE(S(n))$.

6.1 Nondeterministic space vs. deterministic time

The aim is to establish the following theorem.

Theorem 6.4 *Suppose* $S(n) > \log n$ *and that* $S(n)$ *is space constructible, then* $NSPACE(S(n)) \subseteq TIME(2^{O(S(n))})$.

Proof: Let A be a language that can be recognized by a nondeterministic Turing machine N which uses space at most $S(n)$ on inputs of length n. We have to design a deterministic Turing machine that runs in time $2^{O(S(n))}$ which recognizes A.

Assume for simplicity that N has only one worktape, a three letter alphabet, and Q states. Consider the set of configurations of N. Remember that a configuration consists of the state of N, the positions of all its heads and the contents of the worktape. By the argument in the proof of Lemma 3.12 there are at most $|x|QS(|x|)3^{S(|x|)}$ possible configurations that N may visit on input x. Let $G_{x,N}$ be the following directed graph:

The nodes of $G_{x,N}$ are the configurations of N and there is an edge from configuration C_1 to configuration C_2 iff it is possible to go from C_1 to C_2 in one step on input x.

$G_{x,N}$ has one node C_{st} which corresponds to the initial configuration and one or more configurations where N halts with output 1. We now claim that the machine takes value 1 on a given input exactly when there is a path from C_{st} to any of the configurations that end with output 1. This is fairly obvious and the verification is left to the reader. By the above claim $G_{x,N}$ has at most $2^{O(S(|x|))}$ nodes and using the fact that S is space constructible we see that $G_{x,N}$ can be constructed in $2^{O(S(|x|))}$ time. Now it follows from the example in Section 4.2 that in time $2^{O(S(|x|))}$ it is checkable whether any configuration that outputs 1 can be reached from the initial configuration. Since this is equivalent to N accepting x we have proved Theorem 6.4 ▪

We have the following corollary.

Corollary 6.5 $NL \subseteq P$.

Proof: Just insert $S(n) = O(\log n)$ in Theorem 6.4. ▪

6.2 Nondeterministic time vs. deterministic space

This section has only one basic theorem.

Theorem 6.6 $NP \subseteq PSPACE$.

Proof: Remember the characterization of *NP* given in Theorem 5.15, i.e. given $A \in NP$ there is a $B \in P$ and a k such that

$$x \in A \Leftrightarrow \exists y, |y| \leq |x|^k (x, y) \in B.$$

This gives the following algorithm to determine whether $x \in A$

$found = 0$
For $y = 0, 1 \ldots 2^{|x|^k}$ do
  If $(x, y) \in B$ then $found = 1$
od
Write $found$

The algorithm is correct since $found$ will be 1 exactly when there is a short y such that $(x, y) \in B$. To see that the algorithm runs in polynomial space observe that all we need to do is to keep track of y and to do the computation to check whether $(x, y) \in B$. Since this latter computation is polynomial time, we can do it in polynomial space and once we have checked a given y we can erase the computation and use the same space for the next y. ∎

6.3 Deterministic space vs. nondeterministic space

Nondeterministic computation seems very powerful, and it seems for the moment that complexity theory supports this intuition at least in the case when we are focusing on time as the main resource. If, on the other hand, we focus on space it turns out that nondeterminism only helps marginally. This fact is usually referred to as Savitch's theorem and was first proved by W.J. Savitch in 1970.

Theorem 6.7 *If $S(n)$ is space-constructible and $S(n) \geq \log n$, then*

$$NSPACE(S(n)) \subseteq SPACE(O(S^2(n)))$$

.

Proof: Assume that A is accepted by the nondeterministic machine N in space $S(n)$. We will again work with the configurations of N and in fact if you look closely, we solve the same graph problem as we did in the proof of Theorem 6.4. This time however we will be concerned with saving space and thus we will never write down the graph explicitly.

Assume for notational simplicity that N has a unique configuration where it halts with output 1. Let us call this configuration C_{acc}. Let C_1 and C_2 be any two configurations of N and let k be an integer. Then we will be interested in the predicate $GET(C_1, C_2, k, x)$ which we will interpret "On input x it is possible to get from configuration C_1 to configuration C_2 in time $\leq 2^k$ and without being in a configuration which uses more than $S(|x|)$ space." (If we think about the graph in the proof of Theorem 6.4 this can be interpreted as "There is a path of length at most 2^k from node C_1 to node C_2".)

Let C_{st} denote the start configuration of N and recall the argument in the proof of Theorem 5.10 that if a machine has an accepting computation then there is an accepting computation which visits each configuration at most once and, in particular, the running time is bounded by the number of configurations. This implies that there is a constant c such that N accepts an input x iff $GET(C_{st}, C_{acc}, cS(n), x)$ is true. Thus all we have to do is to evaluate this predicate in small space and to achieve this, the following observation will be crucial.

$$GET(C_1, C_2, k, x) = \bigvee (GET(C_1, C, k-1, x) \wedge GET(C, C_2, k-1, x))$$

The $\vee$ is here taken over all possible configurations C of N which uses space less than $S(|x|)$. The reason for the above relation is that if there exists a computational path from C_1 to C_2 of length at most 2^k which never uses more than $S(|x|)$ space then there is a midpoint on this path and the configuration at this midpoint can be used as C. Conversely if there is a C that fulfills the left hand side of the above equation, then the two computations from C_1 to C and from C to C_2 can be concatenated to a computation from C_1 to C_2.

The above equation gives the following recursive algorithm to evaluate the predicate GET.

$GET(C_1, C_2, k, x)$
If $k = 0$ then
    Check whether the next-step function of N allows a transition from C_1 to C_2 on input x in one step and set GET accordingly.
else
    For all configurations C which uses space at most $S(n)$:
        Evaluate $GET(C_1, C, k-1, x)$ and $GET(C, C_2, k-1, x)$.
    If for some C both are true, set GET to true and otherwise to false.
endif

By the above argument $x \in A$ iff $GET(C_1, C_2, cS(|x|), x)$ and thus to prove the theorem we need only calculate the amount of space needed to evaluate GET.

We prove by induction that $GET(C_1, C_2, k, x)$ can be evaluated in space $D(k+1)S(|x|)$ for some constant D. This is clearly to true for $k = 0$ since all that need to be done is to check if one of the constantly many possible next steps that N can do from C_1 will take it into C_2.

To do the induction step let us specify more closely how the above procedure works. We loop over all possible C and to remember which C we are currently working on requires space $dS(n)$ for some constant d. For each C we do two evaluations of GET with the parameter $k-1$. These two evaluations are done sequentially and thus we can first do one of the evaluations, remember the result and then do the other evaluation in the same space. By the induction hypothesis this implies that the computation for a fixed C can be done in space $DkS(n)+1$. Provided that $D > d$ the induction step is complete and thus we have completed the proof of Theorem 6.7. ∎

We have two obvious corollaries of the above theorem.

Corollary 6.8 *$NPSPACE = PSPACE$.*

This explains that *NPSPACE* is not a very famous complexity class. We introduced it for symmetry purposes and now that we have proved that we do not need it, we will forget it.

Corollary 6.9 *$NL \subset PSPACE$.*

Proof: By Theorem 6.7 everything in *NL* can be done in space $O(\log^2 n)$ and thus we get a strict inclusion by Theorem 3.14. ∎

Observe that Corollary 6.9 finishes the proof of Theorem 5.8 as promised before.

By now we have gathered some information about the relations between the complexity classes we have defined. Let us sum up the information in a theorem.

Theorem 6.10 *$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$. The inclusion of* NL *in* PSPACE *is strict.*

It is a sad fact for complexity theory that Theorem 6.10 reflects our total knowledge of the relation between the given complexity-classes.

7 Complete problems

Even though Theorem 6.10 gives the present state of knowledge about the defined complexity classes, there are some important things to be said. The common belief today is that all the given inclusions are strict, but unfortunately we have not yet developed the machinery to prove this. One step on the way is to identify the hardest problems within each complexity class. This serves two purposes. Firstly they will serve as candidates that can be used to prove strict inclusions. Secondly, proving a problem complete will give a good hint that it can probably not be placed in a lower complexity class and thus is a good way to classify a problem. We will start by considering a very famous class of problems; the NP-complete problem.

7.1 NP-complete problems

To identify the hardest problem we need first define the concept of "not harder than". There are a couple of different ways to do this but we will only consider one.

Definition 7.1 *Let A and B be two sets. Then $A \leq_p B$ (read as "A is polynomial time reducible to B") iff there is a polynomial time computable function f such that $x \in A \Leftrightarrow f(x) \in B$.*

Clearly this definition is very close to the Definition 2.21. The only difference is that we require the function f to be computable in polynomial time.

We can now proceed to develop a reduction theory similar to the one described in the end of Section 2.7. Instead of talking about recursive and recursively enumerable sets we will talk about P and NP. Many proofs and theorems are similar.

Theorem 7.2 *If $A \leq_p B$ and $B \in P$, then $A \in P$.*

Proof: Suppose the function f in the definition of $\leq_p$ can be computed in time $O(n^c)$ and that B can be recognized in time $O(n^k)$. Then to check whether a given input x belongs to A just compute $f(x)$ and then check whether $f(x) \in B$. To compute $f(x)$ is done in time $O(|x|^c)$ and from this also follows that $|f(x)| \leq O(|x|^c)$ which in its turn implies that $f(x) \in B$ can be checked in time $O(|x|^{ck})$. Thus the procedure works in polynomial time and we can conclude that $A \in P$. ∎

The definition of *NP*-complete is now very natural having seen the definition of r.e.-complete before.

Definition 7.3 *A set A is* NP*-complete iff*

1. $A \in NP$
2. *If* $B \in NP$ *then* $B \leq_p A$.

By dropping the first condition we get another known concept.

Definition 7.4 *A set A is* NP*-hard iff for all* $B \in NP$, $B \leq_p A$.

Before we continue to prove some problems to be *NP*-complete let us prove a simple theorem.

Theorem 7.5 *If A is* NP*-complete then*

$$P = NP \Leftrightarrow A \in P$$

.

Proof: Clearly if $NP = P$ then $A \in P$ since A by the definition of *NP*-completeness belongs to *NP*.

To see the converse assume that $A \in P$ and take any $B \in NP$. Then by property 2 of being *NP*-complete, $B \leq_p A$ and hence by Theorem 7.2 $B \in P$. But since B was an arbitrary language in *NP* we can conclude that $NP = P$. ∎

With this motivation we are ready to study our first *NP*-complete problem. Let SAT be the set of satisfiable Boolean formulas (as introduced in the example in section 5.1).

Theorem 7.6 *(Cook, 1971) SAT is* NP*-complete.*

Proof: We have already established that $SAT \in NP$ (see the example in section 5.1) and thus we need to establish that $B \in NP$ implies that $B \leq_p SAT$.

Assume that B is recognized by a non-deterministic Turing machine N which has one tape, Q states, runs in time n^c and uses the alphabet $\{0, 1, B\}$. Remember that the computation tableau is a complete description of a computation. We will now construct a Boolean formula such that if it is satisfiable then its satisfying assignment will describe a computation tableau of an accepting computation of N on input x.

The formula has two types of variables:

$$\begin{array}{cc} y_{ijk}, & 1 \le i,j \le n^c,\ k \in \{0,1,B\} \textit{ and} \\ z_{ijl}, & 1 \le i,j \le n^c,\ 1 \le l \le Q. \end{array}$$

The intuitive meaning of the variable will be that $y_{ijk} = 1$ iff the symbol k appears in square j at time i and will take the value 0 otherwise while $z_{ijl} = 1$ iff the head is in square j at time i and the machine at this time is in state q_l. Let us denote the length of x by n.

Clearly the y and z variables code a computation completely and thus all that needs to be done is to make a Boolean formula which is true iff the y and z variables code an accepting computation of N on input x. There are three conditions to take care of.

1. The computation starts with x
2. It is a valid computation.
3. The computation accepts.

Of these three conditions, 1 and 3 are very easy to handle. The condition 1 is equivalent to the following conditions:

- For $1 \le j \le n$ we have $y_{1jk} = 1$ iff $k = x_j$.
- For $n+1 \le j \le n^c$ we have $y_{1jk} = 1$ iff $k = B$.
- $z_{1,j,l} = 0$ except when $j = l = 1$ (assuming that q_1 is the start-state).

The condition 3 is equivalent to $y_{n^c11} = 1$ and $z_{n^c1l} = 1$ i.e. at time n^c we have written a 1 in square 1 and the head is located in square 1 and we have halted (assuming that q_l is the halting state).

To see how to translate condition 2 into a formula we will need some more information.

Definition 7.7 *A computational tableau C is* locally correct *if for every i and j there is some correct computation which have the same contents as C in squares (i',j') for $i \le i' \le i+1$ and $j \le j' \le j+2$.*

That computation is a local phenomena is now formalized as follows:

Lemma 7.8 *A computational tableau describes a legal computation iff it is locally correct.*

We leave the easy verification to the reader.

Armed with this lemma we can now express condition 2 in a suitable way. To determine whether the variables y_{ijk} and z_{ijl} describe as legal computation we only have to check all the local correctness conditions. Whether a given local area is correct is described as a condition on $6Q + 18$ variables and since any condition on K variables can be expressed as a formula of size 2^K we can express each local correctness condition in constant size. The conjunction of all these correctness formulas now takes care of condition 2. The size of the formula is $O(n^{2c})$.

We now claim that the conjunction of the formulas taking care of the conditions 1-3 is satisfiable iff $x \in B$. This is fairly obvious since there is a satisfying assignment iff there is an accepting computation which uses at most space n^c and time n^c of N on input x, which by the definition of N is equivalent to $x \in B$. To conclude the proof of the theorem we need just observe that to construct the formula is clearly polynomial time. ∎

Let us make a couple of observations about the above proof. Firstly the final formula is the conjunction of a number of subformula where each subformula is of constant size. Without increasing the size of the entire formula by more than a constant we write each of the subformulas in conjunctive normal form (i.e. as a conjunction of disjunctions). This puts the entire formula on conjunctive normal form. This implies that satisfiability of formulas on conjunctive normal form is *NP*-complete. Let us call this problem CNF-SAT and we have the following theorem.

Theorem 7.9 *CNF-SAT is* NP*-complete.*

The second observation is that the given proof is almost identical to the proof of Theorem 5.15. If one thinks about this, Theorem 5.15 can be used to give another *NP*-complete problem, namely the existence of a computational tableau with certain conditions. However, we do not feel that this is a natural problem and hence we will not make that argument. There are also striking similarities with the proof of Theorem 2.26. It is just a question of coding a computation in a suitable way.

Having obtained one *NP*-complete problem it turns out to be easy to construct more *NP*-complete problems. The main tool for this is given below.

Theorem 7.10 *If A is* NP*-complete and B satisfies $B \in NP$, $A \leq_p B$, then B is* NP*-complete*

Proof: We have only to check that for any C in *NP* it is true that $C \leq_p B$. Since A is *NP*-complete we know that $C \leq_p A$ and hence there is a polynomial-time computable function f such that

$$x \in C \Leftrightarrow f(x) \in A.$$

By the hypothesis of the theorem there is a polynomial time computable g such that

$$y \in A \Leftrightarrow g(y) \in B.$$

Now it clearly follows that

$$x \in C \Leftrightarrow g(f(x)) \in B$$

and since the composition of two polynomial-time computable functions is polynomial-time computable we have proved $C \leq_p B$ and thus the proof of the theorem is complete. ∎

To put the proof in other words: Polynomial-time reductions are transitive i.e. if we can reduce C to A and A to B then we reduce C to B by composing the reductions.

Clearly Theorem 7.10 is much more useful for proving problems *NP*-complete than the original definition. The reason is that to use Theorem 7.10 we only have to make one reduction while to use the definition we have to make a reduction from any problem in *NP*.

Let 3-SAT be the problem of checking whether a restricted Boolean formula given on conjunctive normal form is satisfiable. The restriction is that there are exactly 3 literals (i.e. variables or negated variables) in each disjunction. Such a formula is called a 3-CNF formula and an example is:

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee x_4)$$

This formula is satisfiable as can be seen from the assignment $x_1 = 1, x_2 = 1, x_3 = 1$ and $x_4 = 0$. We have

Theorem 7.11 *3-SAT is* NP*-complete.*

Proof: We will use Theorem 7.10 and since 3-SAT is clearly in *NP* all that we need to do is to find a polynomial-time reduction from CNF-SAT to 3-SAT.

Thus we need to given a CNF-SAT formula ϕ construct in polynomial time a 3-SAT formula $f(\phi)$ such that ϕ is satisfiable iff $f(\phi)$ is satisfiable. Suppose $\phi = \bigwedge_{i=1}^{m} C_i$ where C_i are disjunctions containing an arbitrary number of literals. We will call C_i a *clause* and let $|C_i|$ denote the number of literals in C_i. We will replace each clause by one or more clauses each containing exactly 3 variables. We have the following cases.

1. $|C_i| = 1$.
2. $|C_i| = 2$.
3. $|C_i| = 3$.
4. $|C_i| > 3$.

Let us take care of the cases one by one. Let $x_i,\ i = 1, 2 \ldots n$ be the variables that appear in ϕ and let y_{ij} denote new variables.

(1.) Suppose $C_i = x_j$, then we replace it by

$$(x_j \vee y_{i1} \vee y_{i2}) \wedge (x_j \vee \bar{y}_{i1} \vee y_{i2}) \wedge (x_j \vee y_{i1} \vee \bar{y}_{i2}) \wedge (x_j \vee \bar{y}_{i1} \vee \bar{y}_{i2})$$

.

(2.) Suppose $C_i = (x_j \vee x_k)$ then we replace it by

$$(x_j \vee x_k \vee y_{i1}) \wedge (x_j \vee x_k \vee \bar{y}_{i1})$$

(3.) We keep C_i as it is.

(4.) Suppose $C_i = \bigvee_{j=1}^{k} u_j$ for some literals u_j we then replace C_i by

$$(u_1 \vee u_2 \vee y_{i1}) \wedge (\bigwedge_{j=1}^{k-4} (\bar{y}_{ij} \vee u_{j+2} \vee y_{i(j+1)})) \wedge (\bar{y}_{i(k-3)} \vee u_{k-1} \vee u_k)$$

The formula we obtain by these substitutions is clearly a 3-CNF formula and it is also obvious that given the original formula it can be constructed in polynomial time. Thus all we need to check is that ϕ is satisfiable precisely when $f(\phi)$ is satisfiable.

First assume that ϕ is satisfiable. We now must find a satisfying assignment for $f(\phi)$. We will give the same values to the x_i and must find values for the y_{ij} to satisfy the formula. The clauses constructed according to rules 1-3 are already satisfied and thus will cause no problem. Look at the clauses constructed under rule 4. Since the corresponding clause C_i in ϕ is satisfied, one of the u_j is true and suppose this is u_{j_0}. Now set $y_{ij} = 1$ for $j \leq j_0 - 2$ and $y_{ij} = 0$ for $j > j_0 - 2$ then it is easy to verify that this assignment satisfies $f(\phi)$.

To prove the converse, suppose that $f(\phi)$ is satisfiable and let $x_i = \alpha_i$ be the assignment to the x variables in this satisfying assignment. We claim that this part of the assignment will satisfy ϕ. For clauses that fall under the rules 1-3 this is not too hard to see. Let us consider case 1. If $C_i = x_j$ and $\alpha_j = 0$ then, no matter what the values of y_{i1} and y_{i2} are, at least one of the clauses is not satisfied.

Now consider the case 4. If C_i was not satisfied then all the literals u_j would be false, but this implies that

$$y_{i1} \wedge (\bigwedge_{j=1}^{k-4} (\bar{y}_{ij} \vee y_{i(j+1)})) \wedge \bar{y}_{i(k-3)}$$

would be satisfied, but this is clearly not possible. Thus the reduction is correct and the proof is complete. ∎

Proving problems *NP*-complete is not the main purpose of these notes but let us at least give one more *NP*-completeness proof. Let *3-dimensional matching (3DM)* be the following problem:

Given a set of triplets (x_i, y_i, z_i), $i = 1, 2 \ldots m$ where $x_i \in X, y_i \in Y$ and $z_i \in Z$ where X, Y and Z are sets of cardinality q. Is there a subset S of q of the triplets such that each element in X, Y and Z appear in exactly one of the triplets in S?

Theorem 7.12 *3DM is* NP*-complete.*

Proof: 3DM is clearly in *NP* since a nondeterministic machine can just nondeterministically pick q of the triplets and then check if each element appears exactly once. To prove 3DM *NP*-complete we will reduce 3-SAT to it. Thus given a 3-CNF formula ϕ we must construct an instance $f(\phi)$ of 3DM such that ϕ is satisfiable iff $f(\phi)$ contains a matching.

Suppose ϕ has n variables and m clauses. We will construct an instance of 3DM with three types of triplets, "variable triplets", "clause triplets"

Figure 8: The variable triplets

and "garbage collecting triplets". The elements of the sets X, Y and Z will be defined as we go along. Let us start by defining the variable triplets. Suppose variable x_i appears (with or without negation) in m_i clauses then we will associate with it the following $2m_i$ triplets.

$$\begin{aligned}
T_i^t &= \{(\bar{u}_i[j], a_i[j], b_i[j]) : 1 \leq j \leq m_i\} \\
T_i^f &= \{(u_i[j], a_i[j+1], b_i[j]) : 1 \leq j < m_i\} \bigcup (u_i[m_i], a_i[1], b_i[m_i])
\end{aligned}$$

The elements $a_i[j]$ and $b_i[j]$ will not appear in any other triplets. As can be seen from Figure 8 this implies that any matching M must contain either all triplets from T_i^f or T_i^t for any i. We will let the choice of which of the two sets to pick correspond to whether the variable x_i is true or false. Each clause C_i will have two special values and three triplets. Suppose $C_i = u_{i_1} \vee u_{i_2} \vee u_{i_3}$ and it is the j_k'th time the variable corresponding to the literal u_{i_k} appears. Then we include the triplets

$$(u_{i_k}[j_k], s[i], t[i]), k = 1, 2, 3.$$

Observe that the u_{i_k} should here be interpreted as literals and thus corresponds to either u_l or $\bar{u}_l$, i.e. these are the same elements as in the variable triplets. The elements $s[i]$ and $t[i]$ will not appear in any other triplets and this implies that in any matching precisely one of the triplets corresponding to each clauses will be included. Observe that we can include a triplet precisely when one of the corresponding literals is true.

We have done the essential part of the construction and all that remains is specify the garbage collecting triplets which will match up the $x_i[j]$ and $\bar{x}_i[j]$ that have not been used. This is done by the following triplets

$$(x_i[j], g_1[k], g_2[k]), 1 \leq i \leq n, 1 \leq j \leq m_i, 1 \leq k \leq 2m$$

$$(\bar{x}_i[j], g_1[k], g_2[k]), 1 \leq i \leq n, 1 \leq j \leq m_i, 1 \leq k \leq 2m$$

This enables us to cover any $2m$ literal-elements which have not been matched by previous triplets.

It is clear from the above description that the set of triplets can contain a matching only if the formula is satisfiable. Suppose on the other hand that the formula is satisfiable. Then make the choice of which T sets to pick based on the satisfying assignment. Then for each clause pick a variable that satisfies it and the corresponding clause triplet. This will cover m of the $3m$ literal-elements. The last $2m$ elements can be covered together with the g elements by the garbage collecting triplets.

Thus there is a matching iff there is a satisfying assignment and since the reduction is straightforward the only thing needed to check that it is polynomial time is to check that we do not have to construct too many triplets. However it is easy to check that there are $6m + 3m + 6m^2$ triplets. This concludes the proof. ∎

There are hundreds of known *NP*-complete problems and many appear in the listing in the final part of the excellent book by Garey and Johnson. It turns out that most problems in *NP* that are not known to be in *P* are *NP*-complete. One notable exception is factoring, another one is graph-isomorphism. Let us however move on and consider problems complete for other classes.

7.2 *PSPACE*-complete problems

The theory of *PSPACE*-complete problems is very similar to that of *NP*-complete problems. The concept of reduction is the same and the basic properties are the same. Of course the problems are different.

Definition 7.13 *A set A is* PSPACE-*complete iff*

1. $A \in PSPACE$.
2. *If* $B \in PSPACE$ *then* $B \leq_p A$

We have an immediate equivalent of Theorem 7.5.

Theorem 7.14 *If A is* PSPACE-*complete then*

$$P = PSPACE \Leftrightarrow A \in P.$$

Proof: If you substitute *PSPACE* for *NP* in the proof of Theorem 7.5 you get a proof of Theorem 7.14. ∎

By a similar argument we get:

Theorem 7.15 *If A is* PSPACE-*complete then*

$$NP = PSPACE \Leftrightarrow A \in NP.$$

One last definition for completeness before we go to business.

Definition 7.16 *A set A is* PSPACE-*hard if for any* $B \in PSPACE$, $B \leq_p A$.

Now let us encounter our first *PSPACE*-complete problem. When dealing with *NP*-complete problems we came across the satisfiability of Boolean formulas. Now we will consider quantified Boolean formulas which looks like:

$$\forall x_1 \exists x_2 \ldots Q x_n \phi(x)$$

where each x_1 can take the value 0 or 1 and ϕ is a normal quantifier free formula and Q is either $\exists$ or $\forall$ depending on whether n is even or odd. Let TQBF be the set of True Quantified Boolean Formulas. We have:

Theorem 7.17 *TQBF is* PSPACE*-complete.*

Proof: Let us first check that TQBF can be recognized in polynomial space. We claim that if the formula has n variables and the size of the description of ϕ is bounded by S then to check whether:

$$\forall x_1 \exists x_2 \ldots Q x_n \phi(x)$$

is true can be done in space $O((n+1)S)$. We prove this by induction and first observe that it is certainly true for $n = 0$. To the induction step we use the observation that the given formula is true iff both

$$\exists x_2 \ldots Q x_n \phi(x)|_{x_1=0}$$

and

$$\exists x_2 \ldots Q x_n \phi(x)|_{x_1=1}$$

are true. These two formulas can be evaluated by induction in space $O(nS)$ and since we can evaluate one and then evaluate the other in the same space while only remembering the value of the first evaluation and which formula to evaluate the claim follows. Of course if the first quantifier is $\exists$ we just need to check that one of the values is true. From this the claim follows and thus $TQBF \in PSPACE$.

Remark 7.18 *By being more careful it is not to hard to see that the evaluation actually can be done in space $O(n+S)$.*

Next we need to take care of the slightly more difficult part of proving that if $B \in PSPACE$ then $B \leq_p TQBF$. Suppose that B is recognized by Turing machine M_B which never uses more space than $|x|^c$ on input x for a given constant c.

We will again use the predicate $GET(C_1, C_2, k, x)$ which means that on input x, M_B will get from configuration C_1 to configuration C_2 in at most 2^k steps and never use more space than $|x|^c$. As before we have

$$GET(C_1, C_2, k, x) = \bigvee_C \left(GET(C_1, C, k-1, x) \wedge GET(C, C_2, k-1, x) \right).$$

With the present formalism it is more convenient to think of the $\vee$ as an existential quantifier and we get

$$GET(C_1, C_2, k, x) = \exists_C \left(GET(C_1, C, k-1, x) \wedge GET(C, C_2, k-1, x) \right).$$

Now we could write the two GETs to the right in the same way but this would be mean trouble since we would then get a formula of exponential size. However there is a way around this by replacing the $\wedge$ by a universal quantifier obtaining.

$$GET(C_1, C_2, k, x) = \exists_C \forall_{(A,B)\in\{(C_1,C),(C,C_2)\}} GET(A, B, k-1, x).$$

Now we only get one copy of GET to expand further and if we continue recursively we get $2k$ quantifiers and a final formula $GET(X, Y, 0, x)$. All that remains to do is to check that it is sufficient to quantify over Boolean variables, rather than the more complicated objects we are currently quantifying over, and that the final application of GET can be written as a Boolean formula.

Both these points are easy and let us just give a rough outline. It is straightforward to encode a configuration as a set of Boolean variables. The $\forall$ quantification is just a binary choice and thus can be represented by a Boolean variable which will take the value 0 if we make the first choice and the 1 if we make the other. Finally, to check whether we can get from one configuration to another in one step is just a simple formula where we list all possible transitions of the Turing machine. We leave the details to the interested reader.

Now since $x \in B$ iff $GET(C_{st}, C_{acc}, d|x|^c, x)$ is true for the appropriate constant d and since we know how to write the latter condition as a quantified Boolean formula we have completed the reduction. ∎

In fact if one writes down the final formula carefully one can write it in CNF, i.e. if we restrict the formula ϕ in TQBF to be a CNF-formula we still obtain a *PSPACE*-complete problem. We call this problem TQBF-CNF.

Theorem 7.19 *TQBF-CNF is* PSPACE-*complete.*

To get other *PSPACE*-complete problems we first state an obvious theorem.

Theorem 7.20 *If A is* PSPACE-*complete and B satisfies $B \in PSPACE$, $A \leq_p B$, then B is* PSPACE-*complete.*

PSPACE-problems are not as abundant as *NP*-complete problems and do not come up in as varying contexts. The main source of *PSPACE*- complete problems outside logic is games. It is a only slight exaggeration to say that to determine who is the winner in most games is *PSPACE*-complete.

The reason that games are this hard is that already quantified Boolean formulas can be viewed as a game between two players, "Exists" and "Forall" in the following way. Given a formula "Exists" chooses the values of all variables which correspond to existential quantifiers and "Forall" chooses the values of all variables which correspond to universal quantifiers. "Exists" wins the game iff the final total assignment satisfies the formula. It is not hard to see that the formula is true iff "Exists" wins the game when both players play optimally.

Of course the *PSPACE*-completeness cannot apply to any usual game like chess, since chess is of a given constant size and hence not very interesting from our point of view. But games that can be generalized to arbitrary size are often *PSPACE*-complete (or hard). Thus for instance to determine who is the winner in a given position of generalized checkers or generalized go is *PSPACE*-hard. We will not get into those games but instead consider a more childish game.

"Geography" is a two-person game where one person starts by giving the name of a geographical place and then the two people alternatingly name geographic places subject to the two conditions that no place is named twice and that each name starts with the same letter that the previous name ended by. The first person not being able to name a place with these two conditions loses.
To get a computational problem out of this game let us generalize.

"Generalized Geography" (GG) is a graph game where two people alternatingly choose nodes in a directed graph. Each node must be a successor of the previous node and no node can be chosen twice. The first person having no choice loses the game. Initially the game starts with a given node.

The computational problem is now: Given a graph, which of the two players has a winning strategy?

Let us first observe that clearly this is a generalization of the geography game where the nodes corresponds to places and there is an edge from A to B if A ends with the same letter B starts with. (On the other hand it is a slightly cheating generalization since the skill in the normal game is to know as many geographic names as possible.)

Theorem 7.21 *Generalized geography is* PSPACE-*complete.*

Proof: It is not hard to verify by normal procedures that GG is in *PSPACE* and thus by Theorem 7.20 we need only to prove that TQBF-CNF can be reduced to GG. We will call the players in the game $\exists$ and $\forall$. Given the

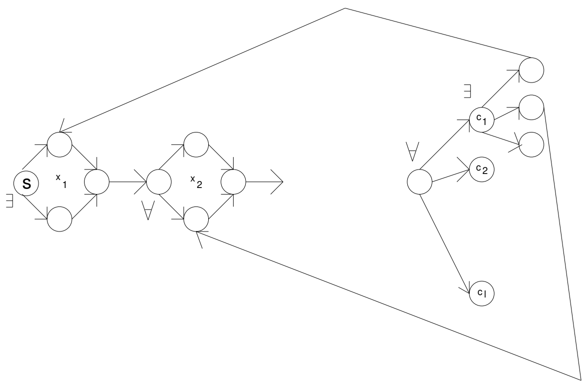

Figure 9: Generalized geography graph

formula

$$\exists x_1 \forall x_2 \exists x_3 [\overbrace{(x_1 \vee \bar{x}_2 \vee x_3)}^{c_1} \wedge \cdots \overbrace{(\quad)}^{c_l}]$$

we construct a graph given in Figure 9. There is a diamond for each variable of the formula, with the last diamond pointing to nodes representing all the clauses of the formula and each clause node pointing to nodes representing the literals in the clause. Finally these nodes are hooked back to the top or the bottom of the diamond for the corresponding variable according to whether the literal is positive or negative. The games starts at the node named S and the $\exists$ and $\forall$ labels in the diagram show whose turn it is to move at each stage.

We can think of $\exists$'s and $\forall$'s choices of how to move through the diamonds as setting the variables (true if the high road is taken and false if the low road is taken). Then $\forall$ gets to pick any clause that he claims to be false, and $\exists$ must pick a literal in that clause which he will claim is true. If $\exists$'s claim is valid, $\forall$ will not be able to move without reusing a node, while if the claim is not true, $\forall$ will be able to move and then $\exists$ will be stuck. Thus we see that $\exists$ has a winning strategy iff the formula is true. Since the reduction clearly is polynomial time we have proved that GG is *PSPACE*-complete. ■

7.3 *P*-complete problems

The question $P = NP$? is of real practical importance since it is a question whether many natural problems can be solved efficiently. The question whether P is equal to L is not of the same practical importance, (although it has a nice connection with parallel computation we have not seen yet) but from a theoretical point of view it is of course of major importance.

Up to this point we have allowed polynomial time for free when we have compared problems. This is clearly not possible when we are considering the question $P = L$? and thus we need a finer reduction concept. The modification is very slight. We just require the reduction-function to be computable in logarithmic space.

Definition 7.22 *Let A and B be two sets. Then $A \leq_L B$ (read as "A is logarithmic space reducible to B") iff there is a function f, computable in logarithmic space, such that $x \in A \Leftrightarrow f(x) \in B$.*

Using this we can now define P-completeness.

Definition 7.23 *A set A is* P*-complete iff*

1. *$A \in P$.*
2. *If $B \in P$ then $B \leq_L A$*

We get the usual theorem.

Theorem 7.24 *If A is* P*-complete then*

$$P = L \Leftrightarrow A \in L.$$

The proof is identical to the other proofs. One small lemma is needed, namely that the composition of two functions in L is in L. We leave this as an exercise. We are now ready to encounter our first P-complete problem. Define a Boolean circuit to be a directed acyclic graph where each node is labeled by either $\wedge$, $\vee$ or $\neg$ and the number of incoming edges is at least two in the first two cases and one in the last. The graph contains sources which are labelled by input variables x_i and one sink which is called the output node. Given values of the inputs to the circuit one can evaluate the circuit in the natural way. An example is given in Figure 10. In this ciruit all edges are directed upwards. Let CVAL be the following problem: Given a circuit and values of the inputs of the circuit. What is the output of the circuit? We have:

Theorem 7.25 *CVAL is* P*-complete.*

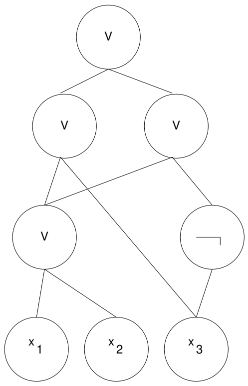

Figure 10: A circuit

Proof: First observe that CVAL belongs to P since it is straightforward to evaluate a circuit once the inputs are given.

Now take any $B \in P$. We need to reduce B to $CVAL$. Assume that B is recognized by a Turing Machine M_B that runs in time at most n^c for inputs of length n. We will again use the concept of a computation tableau. Since we are considering deterministic computation there is a unique computation tableau given the input. The content of each square of the tableau is easily coded by a constant number of Boolean values. We construct a circuit which successively computes these descriptions. The output of the circuit will correspond to the output of the machine, i.e. be the content of the first square at the final timestep.

The content of a given square of the tableau only depends on the contents of the square itself and its two neighboring squares at the previous time step. This means that we can build a constant piece of circuitry that computes the Boolean variables corresponding to the square (i, j) in the computation tableau from the variables corresponding to $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$. Thus to construct a circuit that given the correct input simulates the computation tableau of M_B we just have to copy this piece of circuitry everywhere. To print the description of this circuit on the output tape all we need to remember is the identities of the nodes of the circuits. This can be done in $O(\log n)$ space. Thus in logrithmic space we can construct a circuit and an input to this circuit such that the circuit outputs one iff M_B outputs 1 on input x. Thus we have a correct reduction and the proof is complete. ∎

Several other P-complete problems can be constructed by making logarithmic space reduction from CVAL. We will however not present any more P-complete problems in this section.

7.4 NL-complete problems

The final question we will consider is the $NL = L$? question. Again we have complete problems under L-reductions.

Definition 7.26 *A set A is* NL*-complete iff*

1. *$A \in NL$.*
2. *If $B \in NL$ then $B \leq_L A$*

As before we ge:

Theorem 7.27 *If A is* NL*-complete then*

$$NL = L \Leftrightarrow A \in L.$$

We have already encountered the standard NL-complete problem, namely graph-reachability (GR) i.e. given a directed graph G and two nodes s and t of G, is it possible to find a directed path from s to t.

Theorem 7.28 *Graph-reachability is* NL*-complete.*

Proof: We have more or less already proved the theorem. The fact that $GR \in NL$ was established in Section 5.1.

That the problem is NL-complete was implicitly used in the proof of Theorem 6.4. Let us recall this proof. We started with an arbitrary nondeterministic machine M and an input x to M. We then constructed a graph (of configurations of M) with two special nodes s and t (corresponding to the start configuration and the accepting configuration, respectively) where x was accepted by M iff we could reach t from s. We then observed that graph-reachability could be done in polynomial time and hence $NL \subseteq P$. The first part of this proof is clearly the desired reduction. All we need do is to prove that the reduction can be done in logarithmic space. This is not hard and we leave this to the reader. ∎

8 Constructing more complexity-classes

Let us just briefly mention some more complexity-classes which are very related to the given classes. Before we have pointed out that P is symmetric with respect to complementation i.e. if a set A belongs to P then so does its complement $\bar{A}$. We have also pointed out that this is not true for NP. Thus it is natural to talk about the set of languages whose complement belongs to NP.

Definition 8.1 *A set A belongs to co-NP iff its complement $\bar{A}$ belongs to NP.*

It is in general believed that *co*-NP is not equal to NP. In general for any complexity-class C that is not closed under taking complements, we can define a corresponding complexity-class *co*-C. The only other such class we have encountered is NL.

Definition 8.2 *A set A belongs to co-NL iff its complement $\bar{A}$ belongs to NL.*

It was generally believed that *co*-NL is not equal to NL. Thus it came as a surprise when the following theorem was proved independently by Immerman and Szelepcsényi in 1988.

Theorem 8.3 *If $S(n)$ is space constructible, $S(n) \geq \log n$ and suppose A can be recognized in nondeterministic space $S(n)$, then the complement of A can be recognized in nondeterministic space $O(S(n))$.*

We get the following immediate corollary:

Corollary 8.4 *NL=co-NL.*

Remark 8.5 *Although this theorem was a surprise, one already knew that nondeterminism was not that helpful with regard to space. In particular by Savitch's theorem (Theorem 6.7) we know that whatever can be done in nondeterministic space $S(n)$ can be done in deterministic space $O(S^2(n))$. On the other hand the smallest deterministic time-class that is known to include all things that can be done in nondeterministic time $T(n)$ is essentially $2^{T(n)}$. Thus in spite of the given collapse it is still believed that $NP \neq$ co-NP.*

Proof: For notational convenience we will only prove the corollary. The general case will follow from just substituting $S(n)$ for $\log n$. We will prove that *co*-$NL \subseteq NL$. By symmetry this will imply the equality of the two classes.

Since graph-reachability is complete for *NL*, its complement is complete for *co-NL*. To prove that *co*-$NL \subseteq NL$ we need only prove that graph-non-reachability is in *NL*. In particular we need only to describe a nondeterministic algorithm which works in logarithmic space and given a graph G and two vertices s and t accepts if there is no path from s to t. The idea behind the algorithm is to compute the number of nodes reachable from s. Once we know this number we can verify that t is not reachable by just guessing (and checking) all reachable vertices. Since we cannot guess them all individually, we need to guess them in increasing order. This way we need only remember the number of vertices seen this far and the last one seen.

The number of reachable vertices is computed iteratively. In stage k we compute the number of vertices which are reachable with at most k edges. This is done by at each stage nondeterministically generating all vertices that can be reached in $k-1$ steps. Since we know their number, we know when we have generated all, and thus we can without error decide if a given vertex is reachable in k steps. The complete algorithm now works as follows:

$N_k = 1$
for $k = 1$ to n do
    $newN_k$=0
    for $l = 1$ to n do
        $check = 0$
        for $m = 1$ to n do
            Nondeterministically try to generate a path from s to
            v_m of length at most $k-1$.
            If this is successful then
                $check = check + 1$
                If v_m is connected to v_l (or equal to v_l) then
                    set $newN_k = newN_k + 1$
                    goto next l
                endif
            endif
        next m
        if $check \neq N_k$ reject and stop
    next l

$N_k = newN_k$
next k
$check = 0$
for $m = 1$ to n do
Nondeterminstically try to generate a path from s to v_m of length at most $n - 1$. If this is successful then
$check = check + 1$
If v_m is t reject and stop
endif
next m
if $check = N_k$ accept otherwise reject

We need to prove that it is correct and that it only uses logarithmic space. Let us start with the latter part. The variables used by the program is k, l, m, N_k, $newN_k$ and $check$. It is easy to see that each of them is an nonnegative integer which is at most n and thus we can store these values in space $O(\log n)$. On top of this we need to nondeterministically guess a path of at most a certain length at certain parts of the program. This can be done in logarithmic space by the example in section 5.1 augmented with a simple counter.

Now let us consider correctness. We claim that, unless the algorithm has already halted and rejected, the counter N_k will at stage k give the number of vertices reachable by a path of length at most k from s. We prove this by induction and the base case $k = 0$ is trivial since only s can be reached with 0 edges and N_k is initially 1. For the induction step observe that since the algorithm does not halt and by the induction hypothesis, for each l the algorithm generates all v_m which can be reached in at most $k-1$ steps. Thus it is easy to see that the algorithm decides correctly whether v_l is reachable in at most k steps and thus the new value of N_k will be correct and the induction step is complete.

Finally, for the final loop observe that if in the end $check = N_k$ then we have generated all vertices that are reachable from s with at most $n - 1$ steps (and hence reachable at all) and if t was not one of them we accept correctly. The argument is complete and we have proved Corollary 8.4. ∎

9 Probabilistic computation

From a practical point of view it is sufficient if an algorithm is fast most of the time. On could even relax conditions even further and just ask that the algorithm is correct most of the time.

A key point when reasoning about such algorithms is to make precise what is meant by "most of the time" i.e. we need to introduce some probabilistic assumptions. There are two basic ways to do this:

1. To consider a random input, i.e. to take a probability distribution over the inputs and ask that the algorithm performs well for most inputs.

2. To allow the algorithm to make random choices, and require that the algorithm is fast (correct) for *every* input.

Of course one could also combine the two ways of introducing randomness.

Both approaches give many interesting results, but here we will only study the second approach.

Definition 9.1 *A* probabilistic Turing machine *is a normal deterministic Turing machine equipped with a special coinflipping state. When the machine enters this state it receives a bit which is 0 with probability* $1/2$ *and 1 with probability* $1/2$.

As with nondeterministic Turing machines, a probabilistic Turing machine can do many different computations on a given input. Thus for instance, the output is not uniquely determined, but rather is given by a probability distribution. Also the running time is a random variable and we will say that a probabilistic Turing machine runs in time S if it always halts in time $S(n)$ on every input of length n. Another interesting running time characteristic is the *expected running time.*
We can now define a new complexity class.

Definition 9.2 *A set A belongs to* BPP *iff there is a polynomial time probabilistic Turing machine M such that*

$$\begin{aligned} x \in A \Rightarrow \quad & Pr[M(x) = 1] \geq 2/3 \\ x \notin A \Rightarrow \quad & Pr[M(x) = 1] \leq 1/3 \end{aligned}$$

BPP *is an abbreviation for Bounded Probabilistic Polynomial time.*

Thus the machine M gives at least a reasonable guess of whether an input x belongs to A (We will later see that this guess can be improved). To get the ideas behind these definitions, let us next give an example of a language in BPP not known to be in P.

Example 9.3 *Checking polynomial identities:* Given two polynomials P_1 and P_2 in several variables represented in some convenient way (e.g. as determinants, products or something similar). Do P_1 and P_2 represent the same polynomial? We require that the representation is such that if we are given values of the variables then we can evaluate the polynomials in polynomial time. A typical example would be to investigate whether the equality

$$\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & x_3^{n-1} & \cdots & x_n^{n-1} \end{vmatrix} = \prod_{i>j}(x_i - x_j)$$

is a true identity.

The obvious approach to this problem is to expand the polynomials into a sum of monomials and then compare the expansions term by term. This procedure will in general be quite inefficient since there might be exponentially many monomials (as in the example given). Our probabilistic algorithm will evaluate the two polynomials at randomly chosen points. If the polynomials disagree on one of these points they are different and we will prove that if they agree on all points then they are probably the same polynomial. The algorithm will depend on two extra parameters, d and k. The first parameter is a known upper bound for the degrees of the polynomials in question (in our example we could take $d = \frac{n(n-1)}{2}$) and the second is related to the error probability.

Input P_1 and P_2
For $i = 1, 2$ to k

> Pick random integer values independently for x_1 through x_n in the range $[1, 2nd]$. If $P_1(x) \neq P_2(x)$ conclude that $P_1 \neq P_2$ (answer 0) and stop.

Next i.
Conclude that $P_1 = P_2$ (answer 1).

Clearly iff we answer 0 we are always correct and to see that the algorithm is useful we have to prove that most of the time we are correct even when we answer 1. The key lemma is the following.

Lemma 9.4 *Given a nonzero polynomial, P, in n variables and of degree $\leq d$. The set*

$$Z = \{x \mid 1 \leq x_i \leq R,\ 1 \leq i \leq n\ \wedge\ P(x) = 0\}$$

has cardinality at most dnR^{n-1}.

Proof: We prove the lemma by induction over n. For $n = 1$ the lemma follows from the fact that a polynomial of degree d has at most d zeroes.

For the induction step, let us consider the polynomials Q_j in the variables $x_1 \ldots x_{n-1}$ obtained by substituting j for the variable x_n. Q_j is a polynomial of degree $\leq d$ in $n-1$ variables and thus we could use the induction hypothesis if we knew that Q_j was nonzero. We claim that there are at most d different j such that Q_j is identically zero. To see this take any monomial in P which appears with a nonzero coefficient (assume for the sake of the argument that it is $x_1 x_2 x_n$). Now look at the coefficient of $x_1 x_2$ in Q_j. It is the value at j of a nonzero-polynomial of degree $\leq d-2$. Thus there are at most $d-2$ values of j such that this coefficient is 0 and in general at most d values of j such that Q_j is identically zero.

The set Z splits into the union of sets obtained by fixing the last coordinate to any value in the range 1 to R. When the corresponding polynomial is nonzero, then by the induction hypothesis the cardinality of the set is bounded by $(n-1)dR^{n-2}$ and when the polynomial is zero the cardinality is R^{n-1}. Since there are at most R sets of the first kind and d of the second we get the total estimate

$$R(n-1)dR^{n-2} + dR^{n-1} = ndR^{n-1}$$

and the induction is complete. ∎

Using this lemma we can analyze the algorithm. If P_1 and P_2 represent the same polynomial then we will always answer 1 and we always get the correct answer. When P_1 and P_2 do not represent the same polynomial call an x such that $P_1(x) = P_2(x)$ an *unlucky* x. Thus the algorithms gives the correct answer unless we happen to pick k unlucky x's. By applying the above lemma to $P_1 - P_2$ we see that there are at most $(2dn)^n/2$ unlucky x and thus the probability that we pick one unlucky x is bounded by $\frac{1}{2}$. Since the k x's are independent the probability of them all being unlucky is at most 2^{-k}. Thus if k is reasonably large we get the correct answer with high probability.

All that remains to see that the problem lies in *BPP* is to observe that the algorithm is polynomial time, but this is obvious since the essential step of the algorithm is to evaluate the polynomials and this is polynomial time by assumption.

In the example we saw that if we were willing to run the algorithm longer (i.e. try more random points) then we could make the probability of error arbitrarily small. It is not hard to see that this is true in general.

Theorem 9.5 *A set A belongs to* BPP *iff there is a polynomial time probabilistic Turing machine M such that*

$$x \in A \Rightarrow \quad Pr \quad [M(x) = 1] \geq 1 - 2^{-|x|-2}$$
$$x \notin A \Rightarrow \quad Pr \quad [M(x) = 1] \leq 2^{-|x|-2}$$

Proof: Clearly the above conditions are stronger than our original definition and thus if A satisfies the above condition then it belongs to *BPP*.

We need to prove the converse i.e. that if $A \in BPP$ we can find a machine M which satisfies the above condition. We know by the definition of *BPP* that there is a machine M' such that

$$x \in A \Rightarrow \quad Pr[M'(x) = 1] \geq 2/3$$
$$x \notin A \Rightarrow \quad Pr[M'(x) = 1] \leq 1/3.$$

Now let M be defined by running M', $2(|x| + 3)/\log(9/8) = C$ times with independent random choices and outputting 1 iff M' outputs 1 at least $C/2$ times. We need to verify the claim that this M satisfies the condition in the theorem.

Assume that $x \in A$ and that M' outputs 1 with probability p on input x (we know that $p \geq 2/3$). Then the probability that M does not output 1 is bounded by

$$\sum_{i=0}^{C/2} \binom{C}{i} p^i (1-p)^{C-i}.$$

The ratio of two consecutive terms in this sum is at least $\frac{p}{1-p} \geq \frac{2/3}{1/3} \geq 2$ and thus if the last term is T then the sum is bounded by $\sum_{i=0}^{C/2} 2^{i-C/2} T \leq 2T$. This last term is bounded by

$$2^C (2/3)^{C/2} (1/3)^{C/2} \leq (8/9)^{C/2} \leq 2^{-|x|-3}$$

and thus the first condition of the theorem follows. The second condition is proved in a similar way. ∎

In our example we proved more than needed to establish that the problem in question was in *BPP*. In particular we proved that if the input was in the language the answer was always correct. With this additional restriction we get a new complexity class.

Definition 9.6 *A set A belongs to* R *iff there is a polynomial time probabilistic Turing machine M such that*

$$\begin{aligned} x \in A \Rightarrow \quad & Pr \quad [M(x) = 1] \geq 2/3 \\ x \notin A \Rightarrow \quad & Pr \quad [M(x) = 1] = 0. \end{aligned}$$

Remark 9.7 *I believe that* R *is short for* Random polynomial time*. Hence this class is sometimes also called* RP*.*

While *BPP* is closed under complement, this is not obvious (or known) for *R* and thus we also have a third probabilistic complexity class, *co-R*, the set of languages whose complement lies in *R*. Observe that both *R* and *co-R* are subsets of *BPP*. Our example "Polynomial identities" is a member of *co-R*.

There are not many known examples of problems not known to be in *P* that lie in *BPP*. The main other example is to recognize primes. We will not discuss that algorithm here. However, by quite elaborate methods it is possible to prove that primes belongs to $R \bigcap co\text{-}R$ and for this class we can make a very strong statement.

Theorem 9.8 *A set A belongs to $R \bigcap co\text{-}R$ iff there is probabilistic machine M which runs in expected polynomial time and always decides A correctly.*

Proof: By assumption there is a machine M_1 that outputs 1 with probability at least $2/3$ when the input x is in A and with probability 0 when x is not in A (since $A \in R$). Similarly since $A \in co\text{-}R$ there is a machine M_2 that outputs 1 with probability at least $2/3$ when x is not in A and never when x in A. Both M_1 and M_2 run in polynomial time. Now on input x alternate in running M_1 and M_2 until one of them answers 1. When this happens we know that $x \in A$ if the 1-answer was given by M_1 and we know that $x \notin A$ if it was given by M_2. Each time we run both machines we have probability $2/3$ of getting a decisive answer and hence it follows that the procedure is expected polynomial time. ∎

9.1 Relations to other complexity classes

Let us relate the newly defined complexity classes to our old classes. Clearly any of the defined classes contains P since we can always ignore our possibility to use randomness. We have some non-obvious relations.

Theorem 9.9 *$R \subseteq NP$.*

Proof: We know by the definition of R that if $A \in R$ then there is a machine M such that when $x \in A$ then with probability $\geq 2/3$ M accepts x and when $x \notin A$ there are no accepting computation. But this implies that if we replace the probabilistic choices by non-deterministic choices M accepts x precisely when $x \in A$. ∎

The above theorem immediately yields:

Theorem 9.10 *co-$R \subseteq$ co-NP.*

Our next theorem is also not very surprising.

Theorem 9.11 *Suppose $A \in BPP$ and the machine M that recognizes A runs in time $T(n)$ and uses at most $p(n)$ coins, then A can be recognized by a deterministic machine that runs in time $O(2^{p(n)}T(n))$ and space $O(T(n)+p(n))$.*

Proof: Just run M for all possible $2^{p(n)}$ set of coinsflips and calculate the probability that M accepts. A straightforward implementation gives the given resource bounds. ∎

We have an immediate corollary:

Corollary 9.12 *$BPP \subseteq PSPACE$.*

Apart from these theorems, nothing is known about the relation between our probabilistic classes and our old classes. There is not a great consensus what the true relations are, but many people think it is possible that $P = BPP$.

10 Pseudorandom number generators

In the last section we used random numbers. Without discussing the matter, we assumed that we had access to an unlimited number of perfectly random coins. In practice this might not be the case. One could indeed question whether there are any random phenomena in nature, and thus whether randomness in computation at all makes sense. This is a valid question, but it is mostly philosophical in nature and we will not discuss it. Instead we will take the optimistic attitude that there is randomness, but there is a problem getting enough random numbers into the computer. For the sake of this section we will assume that we only need random bits, where each bit is 0 and 1 with probability 1/2. This is not a severe restriction since random bits can be turned into random numbers in many ways.

The common solution to the problem of not having enough truly random numbers is to have a what is generally called a *pseudorandom number generator* (we will in the future call them *pseudorandom bit generators* since we will be generating bits). This is a function which takes a short truly random string and produces a longer "random looking" string. How the short truly random string (which is called the *seed*) is produced is clearly a problem (it is generally supplied by the user), but we will not concern us with this problem, just assume that somehow we can get a few random bits into the computer.

The main question we will deal with in this section is how to define what we want from a pseudorandom generator and how to construct such a generator. One obvious property is that it should be easy to run and produce something useful, i.e. it should be computable in polynomial time and the output should be longer than the input. Something that has only these two properties is a *bit generator*.

Definition 10.1 *A* bit generator *is a polynomial time computable function that take a binary string as input and on an input of length n produces an output of length $p(n)$ where p is a polynomial such that $p(n) > n$ for all n. For technical reasons we assume that $p(n)$ is strictly increasing with n.*

Note that the definition allows for the output to be of only length $n + 1$ and this does not seem to be much of a generator. We will see later (Theorem 10.8) that this is not a real problem.

The more interesting aspect of pseudorandom bit generators is to try to formalize the "random looking" requirement of the output. Traditionally, this was interpreted as that the output bits passed a small set of standard statistical tests. This is the germ of what today is believed to be the correct definition.

Definition 10.2 *A* statistical test *is a function from binary strings to* $\{0, 1\}$*. Intuitively the output 1 can be interpreted as the string passes the test and the output 0 as failing.*

Note, however that not even all strings produced truly at random will pass a statistical test.

Definition 10.3 *(First attempt) A bit generator* passes *a statistical test* S *if the probability that* S *outputs 1 on a random output of the generator is equal to the probability that* S *outputs 1 on a truly random string.*

Here a random output of the generator is defined as the output on a truly random seed. The tempting definition of pseudorandom generator is now:

Definition 10.4 *(First attempt) A bit generator is* pseudorandom *if it passes all statistical tests.*

A bit generator that passes all statistical tests produces a very random looking output. However the definition is too restrictive and there is no such generator. Take any bit generator G and consider the following statistical test:

$$S_G(x) = \begin{cases} 1 & \text{if } x \text{ can be output by } G \\ 0 & \text{otherwise} \end{cases}$$

First observe that if G stretches strings of length n to strings of length $p(n)$ in time $T(n)$ then S_G can be implemented on strings of length $p(n)$ to run in time $2^n T(n)$ since we just run G on all possible strings of length n and check if one of them equals x.

When we run S_G on the output of G then the result will always be 1. On the other hand when we feed S_G a truly random string then the probability that we get output 1 is at most $1/2$. This follows since there is one output for each seed which implies that there are at most 2^n possible outputs of G of length $p(n)$ (here we use that p is strictly increasing), and since there are $2^{p(n)}$ possible strings and $p(n) \geq n + 1$ at most half of the strings are possible outputs of G.

In practice, if n is large, it is not feasible to compute S_G as described above, since the exponential time needed to try all the seeds is usually too much. Thus somehow this test is "cheating" and we change the definition to take care of this.

Definition 10.5 *(Final attempt) A bit generator is* pseudorandom *if it passes all statistical tests that run in probabilistic polynomial time.*

Remark 10.6 *From the development up to this point polynomial time is the natural requirement on efficient statistical tests. The choice to allow statistical tests to be probabilistic is not clear, but for many reasons (we will not go into them here) it is the better choice. Allowing randomness makes the definition stronger since anything that passes all probabilistic polynomial time statistical tests also pass all deterministic polynomial time statistical tests.*

We have still not overcome all problems with the definitions as can be seen from following miniature version of S_G.

Test s_G

> On input x of length $p(n)$ guess n^2 random seeds of length n and run G on these seeds and output 1 if one of the outputs seen from G is equal to x. Otherwise output 0.

Since G is assumed to be polynomial time, s_G can be implemented in polynomial time. Furthermore, if x is a string that could have been generated by G then there is some small but positive probability that s_g will output 1 while if x cannot be output by G then this probability is 0. By the analysis of S_G this implies that the probability that s_G outputs 1 on a random output of G is different from the probability that it outputs 1 on a random input. As we have defined passing statistical tests this means that G fails the test s_G. This is counterintuitive since for large n the test s_G is very weak. We change the definition to take care of this anomaly.

Definition 10.7 *(Final attempt) Let S be a statistical test and let G be a bit generator. Let a_n be the probability that S outputs 1 on a random output of G of length n and let b_n be the probability that it outputs 1 on a truly random input of length n. G* passes *the statistical test S if for any k there is a N_k such that for all $n > N_k$ it is true that $|a_n - b_n| < n^{-k}$. The probability is taken over the random output of G and the random choices of S.*

In other words the difference of the behavior of the test on the outputs of the generator and random strings goes to 0 faster than the inverse of any polynomial.

Let us first prove that once you have a pseudorandom generator which extends the seed slightly, then we get an arbitrary extension.

Theorem 10.8 *If there is a pseudorandom bit generator G, then for any strictly increasing polynomial p there is a pseudorandom bit generator G' that extends from n bits to $p(n)$ bits.*

Proof: The only problem is that G might not extend the seed sufficiently. By definition G maps n bits to more than n bits. We will assume that G outputs $n+1$ bits since if it outputs more bits we can just ignore them. Note that G remains a pseudorandom bit generator (Prove this!) Now define G' to be G iterated $p(n)-n$ times, i.e. on an input of length n, we first compute G to get a string of length $n+1$, then compute G on this string to get a string of length $n+2$ etc. until we have a string of length $p(n)$. This generator produces a string of the wanted length and it is easy to see that it works in polynomial time. We prove that it is pseudorandom by converting a hypothetical statistical test S which distinguishes the output of G' from random strings to a test which distinguishes the output of G from random strings.

Let a_n be the probability that S outputs 1 on random outputs from G of length $p(n)$ and let b_n be the corresponding probability when the input is truly random. By assumption for some k and infinitely many (for notational convenience we assume this is true for all) n we have $|a_n - b_n| \geq n^{-k}$. Consider the following probability distribution R_i, $0 \leq i \leq p(n)-n$ on strings of length $p(n)$. Start with a truly random string of length $n+i$ and iterate G $p(n)-i-n$ times. Note that R_0 are random outputs of G' while $R_{p(n)-n}$ are truly random strings. Let q_i be the probability that S outputs 1 on distribution R_i. Since $q_0 = a_n$ and $q_{p(n)-n} = b_n$ and $|a_n - b_n| \geq n^{-k}$ there is some i such that $|q_i - q_{i+1}| \geq \frac{1}{n^k p(n)}$. Let us fix this i.

Now consider the following statistical test on strings of length $n+i+1$: Given a string x iterate G $p(n)-n-i-1$ times and run S. If the initial string was random we have produced an element according to R_{i+1} and the probability of getting output 1 is q_{i+1}. On the other hand if the initial string was the output of G on a random string of length $n+i$, then we have produced a string according to R_i and the probability of getting a 1 is q_i. This implies that we have found a way of distinguishing the output for G from random strings and hence we have reached a contradiction since G was supposed to be pseudorandom. Note that the test obviously runs in polynomial time.

This should finish the proof, but the very careful reader will see that there are some minor problems. The proposed test uses two auxiliary parameters $p(n)$ and i. The value $p(n)$ causes no problems since it is the value of a fixed polynomial. However it is not clear how to find i. We sketch how to get around this problem: Let c be a constant. On a given input of length n consider the tests given by different values of i. For each test evaluate the test by picking n^c random inputs according to both distributions. Let i_0 be the value that gives the biggest difference between the two distributions. Now run the test with $i = i_0$ on the given input. It is a tedious (and not that easy) exercise to check that for some c this "universal" test will distinguish the random strings from outputs of G. ∎

Let us next investigate the existence of pseudorandom bit generators.

Theorem 10.9 *If $NP \subseteq BPP$ then there are no pseudorandom generators.*

Proof: Just observe that the test S_G is in NP. Since this test distinguishes the output of G from random bits it should not run in probabilistic polynomial time. ∎

In particular if $P = NP$ there are no pseudorandom generators and thus proving the existence of such generators would prove $P \neq NP$, which we cannot do for the moment. Thus the best we could hope for is to prove that if $P \neq NP$ then there are pseudorandom generators. Also this is probably too much to hope for. The reason is that P vs NP is a question of the worst case behavior of algorithms while the existence of pseudorandom generators is an average case question. This forces us to base the construction of pseudorandom generators on even stronger assumptions.

Definition 10.10 *A function f is a* one-way *function if it is computable in polynomial time and for any probabilistic polynomial time algorithm A the following holds. Choose a random input x of length n and compute $y = f(x)$. If A is given y as input, then the probability that it outputs a z such that $f(z) = y$ goes to 0 faster than the inverse of any polynomial.*

Remark 10.11 *Note that we cannot ask A to actually find the initial x, since in such a case the constant function would be one-way.*

We have:

Theorem 10.12 *If there is a pseudorandom bit generator then there is a one-way function.*

Proof: We claim that the function given by the generator (i.e. from the seed to the output) is one-way. By Theorem 10.8 we can assume the generator expands n bits to $2n$ bits. Assume that the function given by this generator (let us by abuse of notation call the generator as well as the function it computes by G) is not one-way, in other words that there is a k and an A such that A finds an inverse image of a given function value with probability at least n^{-k} (for infinitely many n). Then the following test, S, will distinguish outputs of G from random bits.

> On input x run A. Suppose A outputs y, then if $G(y) = x$ output 1 otherwise output 0.

If x is a truly random string of length $2n$ then the probability that the test S outputs 1 is bounded by the probability that x can be output from G. Since there are 2^{2n} possible strings and at most 2^n outputs from G this probability is bounded by 2^{-n}. On the other hand if x is the output of G then the probability of output 1 is exactly the success probability of A which by assumption is at least n^{-k} (for infinitely many n). Thus this test distinguishes the output from G from random strings contradicting that G is pseudorandom (the test is polynomial time since both A and G are polynomial time). This proves that G is a one-way function. ∎

It was a long standing open question whether the converse of Theorem 10.12 would also be true i.e. if starting from any one-way function it would be possible to construct a pseudorandom bit generator. In 1990 it was proved by Håstad, Impagliazzo, Levin and Luby that this is indeed the case, but their proof is much too complicated for the present set of notes. Instead we prove the following theorem which is due to Yao (the present proof is due to Goldreich and Levin). Let a one-way lengthpreserving permutation be a one-way function which for each n is a 1-1 mapping on strings of length n.

Theorem 10.13 *If there is a one-way lengthpreserving permutation then there is a pseudorandom bit generator.*

Proof: Let f be the one-way lengthpreserving permutation. Let x and r be random strings of length n and let (x, y) be the inner product modulo 2 of the strings x and y (i.e. it is the parity of $\sum_{i=1}^{n} x_i y_i$). Then we claim that the function $g(x, r) = f(x), r, (r, x)$ is a pseudorandom bit generator. It is a bit generator since it expands $2n$ bits to $2n + 1$ bits and is polynomial time computable since f is polynomial time computable. The hard part is to prove that it is pseudorandom. The following lemma of Goldreich and Levin will be crucial.

Lemma 10.14 *Suppose we have a probabilistic polynomial time algorithm A that on input $f(x), r$ computes (x, r) with a probability greater than $\frac{1}{2} + \frac{1}{Q(n)}$ where Q is a polynomial. (Here the probability is taken over a random choice of x and r and the random choices of A). Then there is a probabilistic polynomial time algorithm B that inverts f with probability of success at least $\frac{1}{2Q(n)}$.*

In other words, if f is a one-way function then (x, r) looks random to any probabilistic polynomial time machine which only has the information $f(x), r$.

Let us first see how Theorem 10.13 follows from Lemma 10.14. Suppose g is not pseudorandom and that S is a statistical test which outputs 1 with probability a_n on random bits and b_n on random outputs of g. Suppose without loss of generality that $a_n \geq b_n + n^{-k}$. Now consider the following algorithm for predicting (x, r).

On input $f(x), r$ run S let $b_0 = S(f(x), r, 0)$ and $b_1 = S(f(x), r, 1)$. Now if $b_0 = b_1$ output a random bit and otherwise output i such that $b_i = 1$.

Let $p(x, r, i)$ be the probability that S outputs 1 on $(f(x), r, i)$. Then

$$a_n = 2^{-2n-1} \sum_{x,r,i} p(x, r, i)$$

and

$$b_n = 2^{-2n} \sum_{x,r} p(x, r, (r, x)).$$

Consider the above algorithm on input $f(x), r$. Let $\overline{(r, x)}$ be the complement of (r, x), then the probability that it outputs the correct value for $f(x), r$ is

$$p(x, r, (r, x))(1 - p(x, r, \overline{(r, x)})+$$

$$\frac{1}{2}\left(p(x,r,(r,x))p(x,r,\overline{(r,x)}) + (1-p(x,r,(r,x)))(1-p(x,r,\overline{(r,x)}))\right)$$

which equals $\frac{1}{2}(1+p(x,r,(r,x))-p(x,r,\overline{(r,x)}))$. Hence the total probability of it being correct is $\frac{1}{2}(1+a_n-b_n)$ and now Theorem 10.13 follows from Lemma 10.14. ∎

Next let us prove Lemma 10.14.

Proof: (Lemma 10.14) We give a proof due to Rackoff.

First observe that for at least a fraction $\frac{1}{2Q(n)}$ of the x's, A predicts (r,x) with probability (only over r) at least $\frac{1}{2}+\frac{1}{2Q(n)}$. We will describe a procedure that will be successful with high probability for each such x and this is clearly sufficient.

We compute each bit of x individually. Let e_i be the unit-vector in the i'th dimension. We can ask A about $f(x), e_i$, but there is no reason it will be correct for these inputs. We need to ask about many points, and we will use a small random subspace shifted by a e_i. The set of r's asked will be pairwise independent but we can guess the answers to the entire subspace by guessing the answers on the basis vectors. Let k be a parameter and $\oplus$ be exclusive-or then the algorithm on input y now works as follows:

Pick k random vectors $r_1, r_2, \ldots r_k$ of length n.
For each value of k bits $b_1, b_2 \ldots b_k$ do
    For $i = 1$ to n do
        $count = 0$
        For all non-empty subsets S of $\{1, 2 \ldots k\}$ do
            Ask A about $y, e_i \oplus_{j \in S} r_j$, suppose answer is b.
            Compute $b' = b \oplus_{j\in S} b_j$ and set $count = count + 1 - 2b'$.
        Next S
        Set $x_i = 0$ if $count > 0$ and 1 otherwise.
    Next i.
    If $f(x) = y$ output x, stop
od
Report 'failure'.

Just to avoid confusion observe that *count* is the number of 0-guesses minus the number of 1-guesses and hence we are doing a majority decision. If A runs in time $T(n)$ and f in time $T_1(n)$ then the algorithm runs in time $2^{2k}nT(n)+T_1(n)$ and thus the algorithm is polynomial time if k is $O(\log n)$.

We need to analyze the probability that we find the correct x. We claim that this happens with good probability when $b_i = (r_i, x)$. Let r_S^i be $e_i \oplus_{j\in S} r_j$ and let b_S^i be $b \oplus_{j\in S} b_j$. If A gives the correct answer (i.e. (x, r_S^i)) to y, r_S^i then $x_i = b_S^i$. This implies that we are in pretty good shape since we know that A gives a majority of correct answers and r_S^i are fairly random.

Lemma 10.15 *For $S_1 \neq S_2$ $r_{S_1}^i$ and $r_{S_2}^i$ are independent and uniformly distributed on $\{0,1\}^n$.*

Proof: Suppose $j \in S_1$ but $j \notin S_2$ (if there is no such j we can interchange S_1 and S_2). Now it is easy to see that r_{S_2} is uniformly distributed (its definition is an exclusive-or of several things at least one which is uniformly random) and that for any fixed value of r_{S_2} the existence of r_j in the exclusive-or defining r_{S_1} makes sure it still uniformly distributed. ∎

Now it follows by Lemma 10.15 that the b_S^i are pairwise independent. Suppose for notational convenience that $x_i = 0$ then we know that *check* is a random variable with expected value at least $\frac{2^k-1}{Q(n)}$ and variance at most $2^k - 1$. Now remember, Tchebychev's inequality:

Theorem 10.16 *Let X be a random variable with expected value μ and variance v then the probability that $|X - \mu| \geq \lambda$ is bounded by $\frac{v}{\lambda^2}$.*

Using this with $\lambda = \frac{2^k-1}{Q(n)}$ and $v = 2^k - 1$ we see that x_i takes the incorrect value with probability at most $\frac{Q(n)^2}{2^k-1}$. Now if $2^k - 1 \geq 10nQ(n)^2$ then the probability that x_i does not take the correct value is bounded by $\frac{1}{10n}$. Thus the probability that some x_i is incorrect is bounded by $1/10$. This concludes the proof of Lemma 10.14. ∎

Remark 10.17 *We have now given a generator that extends the input by one bit and we know by Theorem 10.8 that we can get a generator which extends the output arbitrarily. We can take this to be the following very natural generator: Pick x and r randomly and let $b_i = (f^i(x), r)$ where f^i is f iterated i times, for $i = 1, 2, \ldots p(n)$.*

Now that we have studied good generators it is natural to ask what happens if we use these generators to produce the random bits needed by a probabilistic algorithm. Suppose we have a probabilistic machine M which recognizes a BPP-language B and let G be a pseudorandom generator.

Suppose M uses $p(n)$ random bits and that for some small constant ϵ, G extends n^ϵ bits to $p(n)$ bits. The latter can be assumed by Theorem 10.8. Now consider the following statistical test $S_{M,x}$ of a random string r of length $p(n)$:

> Given x run M on input x with random coins r. Answer with the output of M.

We know that when $x \in B$ and r is random then the probability that this test outputs 1 is at least 2/3 while otherwise it is at most 1/3. Since G by assumption passes all statistical tests it is tempting to think that the same is true for outputs of G. This would imply that we would get a theorem similar to Theorem 9.11 saying that B could be recognized in time close to 2^{n^ϵ} since we would only have to try all seeds of G rather than all sets of $p(n)$ coins.

The reason this is not true is that the test has a parameter x which might be hard to find (the parameter M is not a problem since it is of constant size). All is not lost since we could change the statistical test to choose x randomly and then study the behavior of M. Then we could prove that we had a deterministic algorithm that ran in time close to 2^{n^ϵ} and was correct for most inputs. However since we have not studied the concept of being correct for most inputs we will not pursue this approach. Instead we have:

Definition 10.18 *A* non-uniform statistical test *is a probabilistic polynomial time algorithm that on inputs of length n gets an advice a_n which is of polynomial length.*

Remark 10.19 *Note that the advice is the same for all strings of length n. The interested reader might want to prove that the given definition corresponds to polynomial size circuits without any uniformity constraints.*

Definition 10.20 *A pseudorandom generator is* non-uniformly strong *if it passes all non-uniform statistical tests.*

This definition is stronger than the previous definition since we are allowing stronger statistical tests. We will not do so here, but it turns out that the existence of such generators is equivalent to the existence of one-way functions where we allow the inverting function to have an advice. In general all proofs for the uniform case translates to the non-uniform case. In particular Theorem 10.8 remains true. We now finish the discussion with a theorem of Yao.

Theorem 10.21 *If there is a pseudorandom generator which is non-uniformly strong then*

$$BPP \subseteq \bigcap_{\epsilon > 0} DTIME(2^{n^\epsilon}).$$

Proof: The proof is as outlined above. Suppose $B \in BPP$ and that it is recognized by M which uses $p(n)$ coins and runs in time $T_1(n)$ (both these bounds are some polynomials). Let $\delta < \epsilon$ and let G be a non-uniformly strong generator which extends n^δ bits to $p(n)$ bits and runs in time $T_2(n)$ (which also is a polynomial). Now let x be an arbitrary input of length n and consider the above test $S_{M,x}$. This test uses the advice x but since G is non-uniformly strong, it passes this test. This implies that if we replace the coins by a random output of G then we still have essentially the same probability of acceptance. We now just try all the 2^{n^δ} possible seeds for G and take a majority decision. This can be done in time

$$2^{n^\delta}(T_1(n) + T_2(n))$$

and this is $O(2^{n^\epsilon})$. Since both B and ϵ were arbitrary we have proved the theorem. ∎

Thus we have proved that if there are one-way functions in the non-uniform setting then BPP can be simulated in time which is significantly cheaper than exponential time. If one is willing to make stronger assumptions then one can make stronger conclusions. In particular if there is a polynomial time computable function such that inverting this function (in the non-uniform setting, with non-negligible success ration) on inputs of length n requires time 2^{cn} for some small n then $BPP = P$.

11 Parallel computation

The price of processors have dropped remarkably in the last decade and it is now feasible to make computers that have a large number of processors. The most famous multi-processor computer might be the Connection Machine which has $2^{16} = 65536$ processors.

The concept of having many processors working in parallel leads to many interesting theoretical problems. One could phrase the main question as a variant of a traditional mathproblem. Suppose one computer can compute a given function in one million seconds, how long would it take a million computers to compute the same function?

The answer to this question is not known, but it seems like the answer could be anywhere from one second to a million seconds depending on the function. It is an important theoretical problem to identify the computational tasks that can be parallelized in an efficient manner. In this section we will just give the first definitions and show some basic properties.

When many processors cooperate to solve a problem it is of crucial importance how they communicate. In fact it seems like that in practice this is the overshadowing problem to make large scale parallel computation efficient. It is hard to get this fairly practical consideration into the theoretical models in a suitable manner and this complication will usually get lost. We choose here to study the circuit model of computation and as we will see, communication between processors will be ignored. We do not want to argue that the model does not reflect reality, we only want to point out that there is one important aspect missing.

11.1 The circuit model of computation

We have previously briefly discussed the concept of a Boolean circuit. It is a directed acyclic graph with three type of nodes: Input nodes, operation nodes and output nodes. The input nodes are labeled by variable names x_i and the operation nodes are labeled by logical operators. The *inputs* to a node v is the set of nodes w for which (w, v) is an edge.

We will here only allow the operators $\wedge$, $\vee$ and $\neg$. The circuit computes a function $\{0,1\}^n \mapsto \{0,1\}$ in the natural way. (Substitute the value of the i'th coordinate for x_i and then evaluate the nodes by letting each operation node take the value which corresponds to the corresponding operator applied to the inputs of that node.) We will be interested in two parameters of the circuit; its size and depth. The size of a circuit C_n will be denoted by

$|C_n|$ and is equal to the number of nodes it contains while the depth will be denoted by $d(C_n)$ and is the longest directed path from the input to the output. If there is a processor at each node of the circuit then the number of processors is equal to the size of the circuit and the time needed to evaluate the circuit is equal to the depth of the circuit. Thus if we are interested in fast parallel computation it is interesting to construct small circuits with small depth.

The functions we have been considering so far take inputs that are of arbitrary length while a circuit can only take inputs of a given length. The way to resolve this is to let a function be computed by a sequence of circuits $(C_n)_{n=1}^{\infty}$ where C_n computes f on inputs of length n. We will then be interested in the growthrate of the size and depth of C_n as a function of n. In particular we will say that a sequence of circuits is of polynomial size if the growthrate of $|C_n|$ is not more than polynomial in n. Let us now state a theorem that was implicitly proved in Section 7.3.

Theorem 11.1 *If $B \in P$ then B can be recognized by polynomial size circuits.*

Proof: (Outline) In the proof of Theorem 7.25 we saw that given a Turing machine M and an input x we could construct a circuit such that the output of the circuits was equal to the output of M on input x. The circuit constructed the computation tableau of M row by row. If one looks closely at that proof, one discovers that the structure of the circuit only depends on M while x enters as the input of the circuit. In particular, given a language $B \in P$ we take the corresponding Turing machine M_B and given n we can now construct a circuit C_n which will give the same output as M_B on all inputs of length n. The size of this circuit will only be a constant greater than the size of the computation tableau of M_B on inputs of length n. If M_B runs in time $O(n^c)$ then this size will be $O(n^{2c})$ and thus we have constructed circuits for B of polynomial size. ∎

Remark 11.2 *By more efficient constructions it is possible to to give a better simulation of Turing machines and decrease the size of the above circuit to $O(n^c \log n)$.*

One immediate question is whether the converse of the above theorem is true, i.e. that if a function can be computed by polynomial size circuits then is it in fact true that the function lies in P? With the current definitions this is not true. The reason for this is that we have not put any conditions on how to obtain the circuits C_n. To see the problem consider the following language:

$$B = \{x | M_{|x|} \textit{ halts on blank input}\}$$

As we have seen earlier this language is not even recursive. However it has very small circuits since for each length n, either all strings of length n are in B or no string of that length is a member of B. Thus C_n could just be a trivial circuit which either always outputs 0 or 1 depending on whether M_n halts on blank input. How to decide which one to choose is non-recursive but is of no concern in the old definition and the following definition is called for.

Definition 11.3 *A sequence of circuits $(C_n)_{n=1}^{\infty}$ is P (L)-uniform iff there is a Turing machine M, which works in polynomial time (logarithmic space), that on input 1^n prints a description of C_n on its output tape.*

Using this definition we get:

Theorem 11.4 *B can be computed by polynomial size P-uniform circuits iff $B \in P$.*

Proof: (Outline) First just observe that the circuits described in the above proof are P-uniform. They are in fact L-uniform by the proof of Theorem 7.25. This proves one of the implications in the theorem.

To see the reverse implication, suppose that B is recognized by polynomial size P-uniform circuits. Then on input x a Turing machine can first construct the circuit $C_{|x|}$ and then compute its value on input x. The first part is polynomial time by the definition of P-uniform and the second part is easily seen to be polynomial time. ∎

11.2 NC

We can now define our main complexity class of parallel computation.

Definition 11.5 *A set B is in NC^k iff it can be recognized by a family of L-uniform circuits $(C_n)_{n=1}^{\infty}$ where C_n is of polynomial size and $d(C_n) \leq O((\log n)^k)$. Furthermore $NC = \bigcup_{k=1}^{\infty} NC^k$.*

Remark 11.6 *The name* NC *is short for Nick's Class. This is named after Nick Pippenger who was one of the first researchers to study this class.*

Remark 11.7 *Normally one requires even stricter uniformity constraints for NC^1 than L-uniformity. For reasons that go beyond the scope of these notes, this gives a better definition. However to make life easier we will stick with the above definition.*

We can now make an obvious observation.

Theorem 11.8 $NC \subseteq P$.

Proof: This follows immediately from the definition of NC and Theorem 11.4. ∎

From a theoretical standpoint NC is considered as the subset of P which admits ultrafast parallel algorithms (time $O((\log n)^k)$). Some of the algorithms we present will also be efficient in practice and some will not. When we describe how to construct circuits, we will be quite informal and talk in terms of processors doing simple operations. Formally this should of course be replaced by nodes in circuits, but somehow processors seem to go better with the intuition.

Example 11.9 Given two n-bit numbers, compute their sum. This might look straightforward since we can have one processor which takes care of each digit. This will be the basic idea, but we have to do something intelligent with the carries, since if we treat them without thinking, we will need circuits of linear depth. You see the reason for this if you try to add the binary numbers 01111111 and 00000001. The critical point is to discover quickly if you have a carry coming from your right. The process to do this is called *Carry-look-ahead*.

We use one processor for each digit of the two numbers. This processor checks whether that position Generates, Propagates or Stops a carry and marks the position G, P and S accordingly. We can combine this information in a binary tree to see how longer blocks will behave with respect to carries. For instance a block of length two will generate a carry if it looks like GG, GP, GS or PG, it will propagate a carry if it looks like PP and it will stop a carry if it looks like PS, SG, SP or SS. Continuing in this way we can quickly compute whether certain intervals propagate or stop a carry. How to do this might best be seen by an example. Suppose the numbers are 01111011 and 01001010. We get the representation $SGPPGSGP$ and

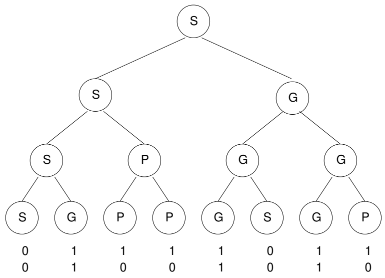

Figure 11: Carry look ahead tree, going up

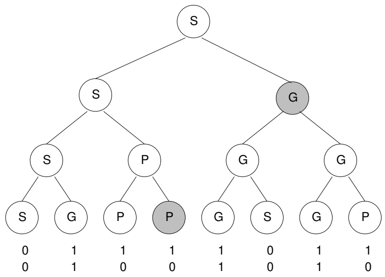

Figure 12: Carry look ahead tree, going down

we build a binary tree (see Figure 11) to find out how longer blocks behave. Now to see if we have a carry in a given position we just have to figure out if all suffices of the string $SGPPGSGP$ generates a carry. It is quite easy to see how this is done. One way to phrase it formally is the following: Suppose you want to know if there is a carry in a given position, start at that position and walk down the tree. Whenever you go right write down what you see coming in from the left to that same node. Finally evaluate the string you get. For instance if you start in position 6 in the given tree, you get the string PG which evaluates to G and thus there is a carry in position 6. One can also view this last step as sending the appropriate values down the tree as indicated in Figure 12. By actually building this tree in the circuit we see that we get a circuit of depth $O(\log n)$ which computes all the carries and since once we know the carries the rest is simple we can conclude that addition belongs to NC^1.

Example 11.10 Given two n-bit numbers, we want to multiply the numbers. It is not hard to see that this can be reduced to adding together n, n-digit numbers (just do the ordinary multiplication algorithm we learned in first grade). Now by the previous example we can add these numbers pairwise in depth $O(\log n)$ to obtain $\frac{n}{2}$ numbers whose sum we want to compute. Adding the numbers pairwise for $\log n$ rounds gives us the answer. This gives a circuit of polynomial size and depth $O((\log n)^2)$. In fact multiplication and addition of n numbers can both be done in depth $O(\log n)$. We leave this as an exercise.

Example 11.11 Given two $n \times n$ matrices, multiply them. Let us suppose the entries are m bit integers. Suppose the given matrices are $A = (a_{ij})$ and $B = (b_{ij})$. Then we want to compute $\sum_{j=1}^{n} a_{ij}b_{jk}$ for all i and k. We have the following algorithm:

1. Compute all the products $a_{ij}b_{jk}$ for all i, j and k.
2. Compute the sums $\sum_{j=1}^{n} a_{ij}b_{jk}$ for all i and k.

If we have $O(m^2n^3)$ processors we can do the first operation in depth $O(\log m)$ (by the exercise extending the multiplication example) while the second can be done with $O(n^3m)$ processors in depth $O(\log nm)$ (using the same exercise). Thus the entire computation uses a polynomial number of processors and $O(\log nm)$ depth.

The problems that seems to be hardest to give a parallel solution to are problems where the natural sequential algorithms are iterative in nature. Examples of such problems are computing integer GCDs, solving linear equations and computing the depth-first search tree of a graph. Of these the linear equation problem can be solved in *NC*, and finding a depth-first search tree is known to be in *RNC* (Random *NC*, i.e. circuits of small depth where you allow random inputs and only require that you have a good probability of finding a depth-first search tree), while for integer GCDs there is not known to be any circuits of sublinear depth. Just to give an example of something nontrivial, let us give as a last example an algorithm to compute the determinant of a matrix which runs in $O((\log n)^2)$ time and uses a polynomial number of processors. We have to assume some facts from linear algebra.

Example 11.12 Given a matrix M, compute its determinant. Let us recall some facts. If λ_i denote the eigenvalues of M, then it is well known that $\prod_{i=1}^{n} \lambda_i = det(M)$. The trace of a matrix M (denoted by $Tr(M)$) is the sum of its diagonal elements, i.e. $Tr(M) = \sum_{i=1}^{n} m_{ii}$, and it is well known that $Tr(M) = \sum_{i=1}^{n} \lambda_i$. Let $s_k = Tr(M^k)$ which equals $\sum_{i=1}^{n} \lambda_i^k$ since the eigenvalues of M^k are λ_i^k. The s_k are easy to compute in parallel since we have already shown how to compute matrix-products and M^k can be computed by $O(\log k)$ matrix-products done in sequence. The characteristic polynomial of M is $det(\lambda I - M) = \lambda^n + \sum_{i=1}^{n} \lambda^{n-i} c_i = c(\lambda)$. It is standard that $c_n = det(-M)$ and $c(\lambda) = \prod_{i=1}^{n}(\lambda - \lambda_i)$. From this it follows that $c_i = \sum_{S:|S|=i}(-1)^i \prod_{j \in S} \lambda_j$ where S is a subset of $\{1, 2, \ldots, n\}$ and $|S|$ is its cardinality. Using this one can prove that

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ s_1 & 2 & 0 & 0 & \ldots & 0 \\ s_2 & s_1 & 3 & 0 & \ldots & 0 \\ s_3 & s_2 & s_1 & 4 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ s_{n-1} & s_{n-2} & s_{n-3} & s_{n-4} & \ldots & n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ \vdots \\ c_n \end{pmatrix} = - \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ \vdots \\ s_n \end{pmatrix}$$

Thus all that remains is to prove that we can solve $Ax = b$ where A is a lower-triangular matrix. If we multiply each row by a suitable number we can assume that all the entries on the diagonal of A is unity. Then A can be written as $I - B$ where B is strictly lower-triangular. Now it is easy to check that $A^{-1} = \sum_{i=0}^{n} B^i$ and thus by some additional matrix-multiplications we can compute the inverse of A and hence we can solve for the c_i and find $c_n = det(-M)$. The number of processors is quite bad but still polynomial, and the depth is $O((\log n)^2)$.

Once we can compute determinants we can do almost all operations in linear algebra. The drawback in practice is that we get fairly large circuits.

11.3 Parallel time vs sequential space

A couple of the examples of problems that we could do in *NC* also appeared as problems doable in small space. This is no coincidence and in fact sequential space and parallel time are quite related as soon as one does not put any other restrictions on the computation.

Theorem 11.13 *Suppose $S(n) \geq \log n$ for all n. If B can be recognized in space $O(S(n))$, then it can be done by circuits of depth $O(S^2(n))$.*

Proof: Suppose B is recognized by M_B which runs in space $O(S(n))$. We will use one processor p_C for each possible configuration C of M_B. There are $2^{O(S(n))}$ configurations and thus we will use many processors, but this is of no concern for us for the moment.

At stage i of the algorithm p_C finds out which configuration C would change to in 2^i computation steps. This is easy for $i = 0$ and in general it is done as follows. After step $i-1$, p_C already knows what configuration C', C transforms to in 2^{i-1} steps. On the other hand $p_{C'}$ knows which configuration C' transforms to in 2^{i-1} steps and this is the desired answer.

Since M_B runs in time $2^{O(S(n))}$, in $O(S(n))$ stages the processor corresponding to the initial configuration will know the result of the computation. Thus the critical parameter is what depth is required to do one stage.

A single stage can be done by having a binary tree of depth $O(S(n))$ which connects each processor to each other processor and selects the processor corresponding to the current information. We leave the details to the reader.

To sum up: We have $O(S(n))$ stages where each stage can be done in depth $O(S(n))$. This gives total depth $O(S^2(n))$ and thus we have proved the theorem. ∎

Corollary 11.14 $L \subseteq NC^2$

Proof: By Theorem 11.13 we know L can be done by circuits of depth $(\log n)^2$. By inspection of the proof we conclude that the circuits are of polynomial size. ∎

There is also a close to converse result to Theorem 11.13. Let S-uniform denote a family of circuits that can be constructed by a Turing machine that runs in space S.

Theorem 11.15 *Suppose $S(n) \geq \log n$ for all n, then if B can be recognized by S-uniform circuits of depth $O(S(n))$, then B can be recognized in space $S(n)$.*

Proof: The idea of the proof is to do a depth first search of the circuit for B.

By duplicating nodes we can assume that the circuit is actually a tree. (One has to check that this does not change the condition of S-uniformity, but it does not) We evaluate the circuit by a depth first search manner. At each point in time we maintain a path in the circuit from the output to an input which has the following properties. Whenever the path goes to the

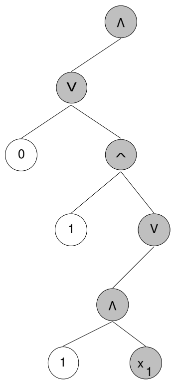

Figure 13: The path at one point in time

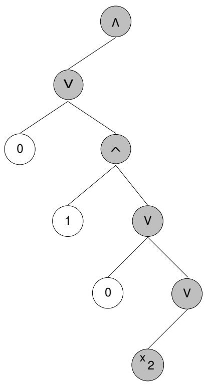

Figure 14: The path at next point in time

left, we require nothing extra while when it turns right we require that we have marked the value of the left input to that node. Also we keep track of what kind of operation we have at each node of the path. We start with the path always going to the left and it is now easy to see that if we always move to the next input to the right it is easy to update the tree. This might best be seen by an example. Suppose our path at one point is given by Figure 13. The active path are the shaded nodes. Assuming that $x_1 = 0$ then at the next time-step a possible path is given by Figure 14. The path is of length $O(S(n))$ and thus can be represented in this space. To update the path we need to be able to find out what the circuit looks like locally, but this can be done in space $O(S(n))$ by the uniformity condition. Thus, we have completed the proof. ∎

Using $S(n) = \log n$ we get the following immediate corollary:

Corollary 11.16 *$NC^1 \subseteq L$.*

With this close connection between L and NC the following theorem is not surprising:

Theorem 11.17 *If A is* P*-complete then*

$$P = NC \Leftrightarrow A \in NC.$$

Proof: The proof is more or less the same as the proof of other theorems of this type, but let us give it anyway. If $P = NC$ then clearly $A \in NC$. On the other hand if $A \in NC$ then we have to construct *NC*-circuits for any function in *P*. Given any $B \in P$ we know by the definition of *P*-complete that there is function f computable in L such that

$$x \in B \Leftrightarrow f(x) \in A.$$

However we know by Corollary 11.14 that f can be computed also in NC^2. Combining this circuit with the *NC*-circuit for A becomes an *NC*-circuit for B. ∎

As a final comment let us note that for one of the most famous problems that seem hard to do in parallel, namely integer GCDs, it is not known that this problem is *P*-complete.

12 Relativized computation

As a tool in understanding computation, one particular way in augmenting the power of a computation has been studied extensively. For definiteness assume that we use the Turing machine model of computation. Let A be a fixed set and give the machine an extra tape, called the *query tape*. On this tape the machine can write a string x and then enter a special state called the *query state*. In one time-step the query tape now changes content. The new value will be 1 if $x \in A$ and 0 otherwise. Thus the machine is allowed to ask questions about the set A and very inexpensively obtain correct answers.

The set A, which is called the *oracle set* should be thought of as a difficult set, since otherwise the machine could have answered the questions itself at only a slightly higher cost. The computation is said to take place *relative* to the oracle A (and hence the title relativized computation). A Turing machine M with an oracle A is usually denoted M^A to avoid confusion.

Now it is natural to define P^A as the set of languages that can be recognized in polynomial time by Turing machines with oracle A. In a similar way all the other complexity classes can be defined. One word of caution. We will count the part of the query tape used as part of the work-tape of the machine and hence this should be bounded when we are looking at space bounded classes. This definition is not standard when dealing with L and NL, but we will not consider those classes here. Instead we will only consider

$$P^A, NP^A, BPP^A \text{ and } PSPACE^A.$$

The reason this concept is interesting is that almost all proofs that are known remain true if we allow all machines involved in the proof have access to the same oracle. In particular this is the case for all proofs given in these notes upto this point. Let us state some theorems that follow (the reader is encouraged to go back and check the proofs).

Theorem 12.1 *For all oracles A,*

$$P^A \subseteq NP^A \subseteq PSPACE^A.$$

Theorem 12.2 *For all oracles A,*

$$P^A \subseteq BPP^A \subseteq PSPACE^A.$$

The idea is that if $P \subset NP$ (i.e. that the inclusion would be strict) has an "easy" proof then $P^A \subset NP^A$ would be true for all oracles A. However this is not the case:

Theorem 12.3 *If A is a $PSPACE$-complete set then*

$$P^A = NP^A = BPP^A = PSPACE^A = PSPACE.$$

Proof: It is sufficient to prove that $PSPACE \subseteq P^A$ and that $PSPACE^A \subseteq PSPACE$.

For the first part let B be anything in $PSPACE$. Since A is $PSPACE$-complete we have $B \leq_p A$ i.e. there is a polynomial time computable function f such that $x \in B \Leftrightarrow f(x) \in A$. But this makes B easy to recognize for a machine with oracle A. On input x it just computes $f(x)$, writes this on the oracle tape, reads the answer from the oracle and outputs this as its own answer. Thus $B \in P^A$ and we conclude that $PSPACE \subseteq P^A$.

For the second part, suppose we are given a machine M^A that recognizes some language in $PSPACE^A$. We have to convert this into an ordinary $PSPACE$-machine which recognizes the same language. Essentially we have to get rid of A. But since A is in $PSPACE$ this is not too difficult. Build a subroutine S which takes an input x and outputs 1 if $x \in A$ and 0 otherwise. This subroutine can be made to run in polynomial space. Now modify M^A, such that instead of entering the query-state it runs S. By definition the result is the same, and it is easy to see that this modified machine also runs in polynomial space. ∎

Theorem 12.3 rules out the possibility of an easy proof that $P \neq NP$. This might raise in a more serious way (at least it seems) the possibility that $P = NP$. However, oracles will not support this:

Theorem 12.4 *There is an oracle B such that $P^B \neq NP^B$.*

Proof: The oracle B will not be as natural as the oracle A given above and we will construct it piece by piece. Together with B we will also define a language $L(B)$ which for all B will be in NP^B, but we will cleverly construct B such that it is not in P^B.

Definition 12.5 *Let $L(B)$ be a language which only contains strings which solely consists of 1's (such a language is called a* unary *language). The string of n 1's is in $L(B)$ if and only if there is at least one string x of length n such that $x \in B$.*

First observe that for any oracle B, $L(B)$ is in NP^B. Formally $L(B)$ is recognized by the following algorithm.

1. If there is a '0' in the input reject and stop.

2. Nondeterministically write down a query to the oracle of the same length as the input. If the oracles answers 1 accept otherwise reject.

To verify that this algorithm is correct is left to the reader.

Next we will have to define B such that $L(B)$ is not in P^B. Let M_i^B be an enumeration of all oracle machines that run in polynomial time. This is a slightly subtle point since whether an oracle Turing machine runs in polynomial time depends on the oracle and we have not yet decided what the oracle should be. This is no real problem and we get around it as follows: Assume that M_i^B is an enumeration of all Turing machines which has the property that each machine appears an infinite number of times . Equip M_i^B with a stop-watch such that if it has not halted in $i|x|^i$ steps on input x, it automatically halts and outputs 1. Now all sets recognized by a polynomial time machine is recognized by some M_i^B (we need to repeat each machine infinitely many times since we do not know for which i it is true that it runs in time in^i. We will now go through an infinite number of stages. In stage i we determine a little bit more of the oracle B to make sure that M_i^B does not recognize $L(B)$. Let a string be *undetermined* if we have not yet decided whether it will be in B.

$n_0 = 1$
for $i = 1$ to ∞ do
    make n_i the smallest number bigger than n_{i-1} such that $2^{n_i} > in_i^i$
    and such that no string of length n_i has been determined.
    Run M_i^B on input 1^{n_i}. Whenever the machine asks about an
    undetermined string, fix that string not to be in B
    If M_i^B accepts the input then
        Make sure that no string of length n_i is in the oracle set.
    else
        Put one undetermined string of length n_i in the oracle set.
    endif
next i
fix all undetermined strings not to be in B.

For the constructed B, M_i^B will not accept $L(B)$ since it will make an error on 1^{n_i}. Hence we need only check that the construction is not contradictory. The only nonobvious point is that when needed there exists an undetermined string of length n_i However, since M_i^B on input 1^{n_i} only runs for time in_i^i and hence it can only ask this many questions. Thus only this many new strings can be determined during stage i and since there were no determined string of length n_i when stage i started and $2^{n_i} > in_i^i$ there is an undetermined string that can be put into B. ∎

It turns out that also all the other questions can be relativized in the possible way. Let us next take NP versus $PSPACE$.

Theorem 12.6 *There is an oracle C such that $NP^C \neq PSPACE^C$.*

Proof: This proof will very much follow the same line as the last proof. Let us start by defining the language.

Definition 12.7 *Let $L_{\oplus}(C)$ be a unary language such that $1^n \in L_{\oplus}(C)$ iff there is an odd number of strings of length n in C.*

First observe that for any oracle C, $L_{\oplus}(C)$ is in $PSPACE^C$. The algorithm just asks all questions of length n and keeps a counter to compute the parity of the number of strings in the oracle. We will now construct C such to make sure $L_{\oplus}(C)$ is not in NP^C.

Using the same argument as in the last proof there is an enumeration N_1^C, $N_2^C, \ldots$ of all polynomial time nondeterministic oracle machines where N_i^C runs in time at most in^i. We now construct C in stages:

$n_0 = 1$
for $i = 1$ to ∞ do
  Make n_i the smallest number bigger than n_{i-1} such that $2^{n_i} > in_i^i$
  and such that no string of length n_i has been determined.
  Consider N_i^C on input 1^{n_i}. If there is some setting of
  undetermined strings to make N_i^C accept then
    Make such a setting, by fixing at most in_i^i strings, fix the
    remaining strings of length n_i to make sure that an even number
    of strings of length n_i are in C.
  else
    Fix strings to make sure that an odd number of strings of length
    n_i are in C.
  endif
Fix all undetermined strings not to be in C.
next i

Again by construction for this oracle $L_{\oplus}(C)$ is not in NP^C. The construction can be seen to be correct by more or less the same reasoning as the last construction. Please observe that if N_i^C accepts an input then it is sufficient to fix the answers of the questions on one accepting computation path and hence it is sufficient to fix in_i^i strings in the first case. ∎

Next we have:

Theorem 12.8 *There is an oracle D such that $BPP^D \not\subseteq NP^D$.*

Proof: We proceed as usual.

Definition 12.9 *Let $L_{maj}(D)$ be a unary language such that $1^n \in L_{maj}(D)$ if a majority of the strings of length n is in D.*

This language is not always in BPP^D. However, if we make sure that for each n, at least 60% or at most 40% of the strings is in the oracle set, then a simple sampling algorithm will work. This extra condition means that we have to be slightly careful in the oracle construction, but there is no real problem. We again give an algorithm to determine the oracle:

```
n_0 = 1
for i = 1 to ∞ do
      Make n_i the smallest number bigger than n_{i-1} such that
      2^{n_i} > 10 · in_i^i and such that no string of length n_i has been determined.
      Fix all undetermined strings of length less than n_i not to be in D.
      Consider N_i^D on input 1^{n_i}. If there is some setting of
      undetermined strings to make N_i^C accept then
         Make such a setting, by fixing at most in_i^i strings and fix the
         remaining strings of length n_i not to be in D.
      else
         Put all undetermined strings of length n_i into D.
      endif
next i
```

The verification that this construction is correct is similar to the previous verifications. The reason to put all undetermined strings of length at most n_i out of the oracle is to make sure that for n's which are not chosen to be one of the n_i's it is also true that the number of strings of length n in the oracle is not close to half of all strings of length n. The condition that $2^{n_i} \geq 10 \cdot in_i^i$ make sure that this is true for all n with $n_i \leq n < n_{i+1}$. ∎

Our last oracle construction will be:

Theorem 12.10 *There is an oracle E such that $NP^E \not\subseteq BPP^E$.*

Proof: We will use the same language as we used in the proof that there was an oracle B such that $NP^B \neq P^B$. Remember that $L(E)$ is a unary language such that $1^n \in L(E)$ iff there is some string of length n in E. We now construct E to make sure it is not in BPP^E. This time let M_i^E be an enumeration of probabilistic Turing machines. Here there is a slight problem that M_i^E might not define a correct machine in that the probability of acceptance is not bounded away from 1/2 for some inputs. However, this is only to our advantage since this means this machine will not accept any BPP-language, and we do not have to worry that it might accept $L(E)$. We now construct E in stages as follows:

$n_0 = 1$
for $i = 1$ to ∞ do
    Make n_i the smallest number bigger than n_{i-1} such that
    $2^{n_i} > 10 \cdot in_i^i$ and such that no string of length n_i has been
    determined.
    Run M_i^E on input 1^{n_i}. Whenever the machine asks about a string
    which is not determined, pretend that this string is not in E. Let p
    be the probability that M_i^B accepts under these conditions.
    If $p \geq 1/2$ then
        Fix all strings M_i^E could possibly ask about not to be in E. Also
        fix all other strings of length n_i not to be in E.
    else
        Find one string of length n_i such that the probability that this
        string is asked by M_i^E is at most 1/10 and put this into E. Fix all
        other strings M_i^E might possibly look at not to be in E
    endif
next i
fix all undetermined strings not to be in E.

Here there are some details to check. If $p \geq 1/2$ then this is actually the correct probability of acceptance since we eventually fix all the strings not to appear in E. In this case $1^{n_i} \notin L(E)$ while the probability that M_i^E accepts 1^{n_i} is at least 1/2 and thus M_i^E does not recognize $L(E)$ in the BPP sense. On the other hand if $p < 1/2$ then the final oracle does not agree with the simulation. However since the probability of finding out the difference is bounded by $1/10$, the acceptance probability remains below 0.6. Since in this case, $1^{n_i} \in L(E)$, also in this case M_i^E fails to recognize $L(E)$.

We need also check that there is a suitable string which is asked with probability at most $1/10$. Since the running time of M_i^E on input 1^{n_i} is bounded by in_i^i it does not ask more than this number of questions. If $PR(x)$ is the probability that string x is asked then

$$\sum_{|x|=n_i} PR(x) \leq in_i^i$$

and since $2^{n_i} > 10 \cdot in_i^i$ there is some x with $PR(x) < 1/10$. The proof is complete. ∎

We have now established that all the unknown inclusion properties of our main complexity classes can be relativized in different directions. The only information this gives is that the true inclusions can not be proved with methods that relativize. In principle, methods that do not look very detailed at the computation will relativize. In particular when you treat the computation as a black box which just takes an input and then produces an output (after a certain number of steps). Thus, the main lesson to learn from this section is that to establish the true relations of our main complexity classes, we have to look in a very detailed way at computation.

There are a few results in complexity theory which do not relativize. One of them (IP=PSPACE) is given in Chapter 13.

13 Interactive proofs

One motivation for *NP* is to capture the notion of "efficient provability". If $A \in NP$ and $x \in A$ then there is a short proof of this fact (the non-deterministic choices of the algorithm which recognizes A) which can be verified efficiently. By the definition of *NP* all proofs are correct and an all powerful prover can always convince a polynomial time bounded verifier of a correct *NP*-statement. As we did with regards to ordinary computation we can introduce randomness and decrease the requirements. A proof will be a discussion (interaction) between an all powerful prover and a probabilistic polynomial time verifier. Before we make a formal definition let us give an example.

Example 13.1 Given two graphs G_1 and G_2 both on n vertices. G_1 and G_2 are said to be *isomorphic* iff there is a permutation π of the vertices such that (i, j) is an edge in G_1 iff $(\pi(i), \pi(j))$ is an edge in G_2. In other words there is a relabeling of the vertices to make the two graphs identical. This problem is in *NP* since one can just guess the permutation. On the other hand it is not known to be in *P* (or *co-NP*) nor known to be *NP*-complete. Now consider the following protocol for proving that two graphs are not isomorphic.

For $m = 1$ to k:
    The verifier chooses a random i (1 or 2) and sends a graph H which is a random permutation of G_i to the prover.
    The prover responds j.
    The verifier rejects and halts if $i \neq j$
next m
The verifier accepts.

In other words the prover tries to guess which graph the verifier started with and the verifier accepts if he always guesses correctly. Now suppose that G_1 and G_2 are not isomorphic. Then H is isomorphic only to G_i and the all powerful prover can tell the value of i and always answer correctly. On the other hand if G_1 and G_2 are isomorphic then, independent of the value of i, the graph H is a random graph isomorphic to both G_1 and G_2. Thus there is no way the prover can distinguish the two cases and thus if he tries to answer he will each time fail with probability 1/2. Thus the probability that he can incorrectly make the verifier accept is 2^{-k} which is very small if k is large. Thus, for all practical purposes if $k = 100$ and the prover always answer correctly the graph will be non-isomorphic.

A discussion (or interaction) of the type described in the example will be called an *interactive proof*. Let us formalize the properties wanted.

Definition 13.2 *A language A admits an interactive proof iff there is an interaction between a probabilistic polynomial time verifier V and an all powerful prover P such that:*

1. *(Completeness) If $x \in A$ then the probability (over V's random choices) that V accepts is at least* $2/3$.
2. *(Soundness) If $x \notin A$ then no matter what the prover does the probability (over V's random choices) that V accepts is at most* $1/3$.

Definition 13.3 *The complexity class* IP *is the set of languages that admit an interactive proof.*

The number of exchanges of messages might depend on the length of the input, but since we want the entire process to be polynomial time, we limit this to be a polynomial number in the length of the input.

Interactive proofs were defined by Goldwasser, Micali and Rackoff in 1985. A different definition that was later proved to give the same class of languages was given independently by Babai around the same time. Interactive proofs attracted a lot of attention in the end of the 1980's and we will only touch on the highlights of this theory. Let us first state an equivalent of Theorem 9.5.

Theorem 13.4 *If $A \in IP$ then there is an interaction between a probabilistic polynomial time verifier V and an all powerful prover P such that:*

1. *If $x \in A$ then the probability (over V's random choices) that V accepts is at least* $1 - 2^{-|x|}$.
2. *If $x \notin A$ then no matter what the prover does the probability (over V's random choices) that V accepts is at most* $2^{-|x|}$.

Proof: (Outline) The proof is very similar to the proof of Theorem 9.5. We just run many protocols in many times and make a majority decision in the end. We leave the details to the reader. ∎

A far less obvious fact is that one can in fact obtain perfect completeness (i.e. when $x \in A$ then the probability that V accepts is 1). Proving this would take us too far and we omit this theorem.

The first couple of years, one of the main drawbacks of the theory of interactive proofs was the small number of languages that were not in *NP* that admitted interactive proofs. This was dramatically changed in December 1989 when work of Nisan, Fortnow, Karloff, Lund and finally Shamir led to the following remarkable theorem:

Theorem 13.5 $IP = PSPACE$.

Proof: (Outline) The fact that $IP \subseteq PSPACE$ was established quite early in the theory of interactive proofs. A formal proof is slightly cumbersome (but not really hard) and hence let us only give an outline. Suppose $A \in IP$ and the interaction that recognizes A contains k pairs of messages. We denote the ith prover message by p_i and the ith verifier by v_i and assume that the prover sends the first message in each round. Now let α be any partial conversation consisting of the first s messages for some s and let $Pr(x, \alpha)$ be the probability that V accepts given that the initial conversation is α and that P plays optimally in the future and that V follows his protocol. Our goal is to compute $Pr(x, e)$ where e is the empty string, since this number is at least $2/3$ when $x \in A$ and less then $1/3$ otherwise. Now if the last message in α is by the verifier then

$$Pr(x, \alpha) = E\left((x, \alpha v_i)\right)$$

where E is expected value over the verifier message v_i. On the other hand if the next message is by the prover then

$$Pr(x, \alpha) = \max\left((x, \alpha p_i)\right)$$

where the maximum is taken over all messages p_i. Finally when α is a full conversation then $Pr(x, \alpha)$ is 1 iff the verifier would have accepted after the conversation α and 0 otherwise. By assumption this can be computed in polynomial time. Using these equations it is easy to give an algorithm that proceeds in a depth first search fashion and evaluates $Pr(x, e)$ in polynomial space.

This inclusion was no surprise since $PSPACE$ is a big complexity class. It was the reverse computation that was the big surprise.

To prove that $PSPACE \subseteq IP$ we need "only" give an interactive proof which recognizes TQBF which was proved *PSPACE*-complete in Theorem 7.17. We only give an outline of the argument.

In fact we will use that determining the truth of the special type of quantified Boolean formulas constructed in the proof of Theorem 7.17 is *PSPACE*-complete. Let us recall part of this proof. We wanted to construct a formula $GET(C_1, C_2, k)$ that said that the Turing machine could get from configuration C_1 to configuration C_2 in 2^k steps. This formula was constructed recursively using:

$$GET(C_1, C_2, k, x) = \exists_C \forall_{(A,B) \in \{(C_1,C),(C,C_2)\}} GET(A, B, k-1, x).$$

Now encode the $\forall$ quantifier as a Boolean variable x_1 and rewrite the formula to the following.

$$GET(C_1, C_2, k, x) = \exists_C \forall_{x_1} \exists_{(A,B)}$$

$$(x_1 \to ((A = C_1) \wedge (B = C))) \wedge (\bar{x}_1 \to ((A = C) \wedge (B = C_2))) \wedge GET(A, B, k-1, x).$$

Now assume that each configuration consists of n Boolean variables and that initially $k = n$. In reality they are both polynomial in n but this is of no importance. It is not difficult to write $(x_1 \Rightarrow ((A = C_1) \wedge (B = C))) \wedge (\bar{x}_1 \Rightarrow ((A = C) \wedge (B = C_2)))$ as a CNF-formula with n clauses and each clause is of polynomial size. Furthermore note that each variable describing C_1 and C_2 does not appear in $GET(A, B, k-1)$. When we iterate the above construction it will be true that no variable in any quantifier will used inside more than 3 other quantifiers. Let us also note that $GET(Y, Z, 0)$ can be done by a CNF-formula with $O(n)$ clauses of constant size. To summarize the discussion the formula has the following properties.

- It has $3n$ quantifiers which appear in blocks of the form $\exists\forall\exists$ where the $\exists$ quantifiers quantify over n variables and $2n$ variables respectively and the $\forall$ quantify over one variable.

- Each variable is used only inside at most one block of following quantifiers.

- All formulas between quantifiers and after the last quantifier are CNF-formulas with $O(n)$ clauses of constant size.

Now take this formula and replace all $\exists$ by $\sum$ and $\forall$ by $\prod$. Here the sums and products extends over all variables that was originally in the scope of the quantifier. Also replace $\wedge$ by $\times$ and $\vee$ by $+$. Finally for a variable replace $\bar{x}$ by $1-x$. Using this replacement the formula is now turned into an expression which evaluates to an integer. It is not difficult to see that this integer is 0 iff the original formula was false (prove this by induction). We will show how the prover can convince the verifier with high probability that this integer I is not 0.

First observe that I is bounded by $2^{O(n2^n)}$. This is true since the value of the final CNF-formula is at most c^n and each $\sum$ only multiplies the value by 2^n while each $\prod$ only squares the value (remember that there is only one variable in each $\prod$). The following lemma follows from the prime number theorem (the reader is asked to take it on faith).

Lemma 13.6 *For $c < 1$ and $x > X_c$, the product of all primes less than x is $\geq e^{cx}$ where e is the base of the natural logarithm ≈ 2.718.*

This lemma implies that there is some prime p, $n^4 \leq p \leq O(2^n)$ such that $I \not\equiv 0$ modulo p. To see this observe that if $I > 0$ and it is divisible by a set of primes then it is at least the product of the primes. The prover starts by giving this p together with I $(mod\ p)$ (which is not 0).

Remark 13.7 *In fact if one is more careful one can make $I = 1$ when the formula is true. This implies that one can use a small prime. This will make the proof slightly more efficient, but this is of no major concern for the moment.*

Now consider the outermost quantified variable. Let us call it x_1 and suppose it is part of an $\exists$ quantifier (i.e. now we are summing over its two values). Keep this variable free and evaluate the entire expression $mod\ p$ with its sums and products. Naturally the result is a polynomial $P(x_1)$ and by the conditions of the formula it is of degree $O(n)$. Here we need both that intermediate pieces of the formula are simple CNF-formula and that the usage of each variable is very limited. The prover now gives this formal polynomial (mod (p)) to V. This can be done since there are $O(n)$ coefficients each which can be specified with $O(n)$ bits. The verifier verifies that $P(0) + P(1) \equiv I$ $(mod\ p)$, and responds with a random integer n_1 where n_1 is chosen randomly among $1, 2 \ldots p-1$. The task for the prover is now to prove that $P(n_1)$ is the value of the algebraic when n_1 is substituted for x_1. The resulting algebraic expression has one quantified variable less and we can now attack the next variable. Once all the variables have been eliminated the verifier can himself evaluate the remaining polynomial and if it equals the value claimed by the prover he accepts and otherwise rejects.

Let us sketch why this protocol is correct. When the formula is true there is really no complications since the prover all the time is claiming correct statements and thus the verifier will accept with probability 1. Note that there is really no difference between the $\forall$-variables and the $\exists$-variables, we only need the assumption on the structure of the formula to make the degree of the polynomial P small.

Suppose on the other hand that the formula is false. In particular $I = 0$ and the first value claimed by the prover for $I \ (mod\ p)$ is incorrect and hence also the first polynomial P is not correct (since it takes an incorrect value for either 0 or 1). Suppose the true polynomial is Q. Let us say that n_1 is *lucky* for the prover if $P(n_1) \equiv Q(n_1) \ (mod\ p)$. If the prover is lucky once then he starts claiming correct statements and thus he will be able to convince the verifier. On the other hand if he is never lucky then he will be forced to continue lying and the verifier will expose him in the end. Since $P - Q$ is a nonzero polynomial of degree $O(n)$ it has at most $O(n)$ zeroes. This implies that the probability that the prover is lucky at a single point is $O(n/p) \leq O(n^{-3})$. Since there are only $O(n^2)$ variables, the probability that he is ever lucky is $O(n^{-1})$. Thus with probability $1 - O(n^{-1})$ the verifier will reject and the protocol is correct.

■

To give a little bit perspective of this proof, let us give an example to show how it works.

Example 13.8 For simplicity let us work with a formula on normal TQBF-CNF form and in particular, consider

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_4).$$

This formulas is true since if we put $x_1 = 0$ and $x_3 = 1$ both clauses are satisfied. It does not matter what happens with the other variables. The formula is turned into the following arithmetical expression:

$$\sum_{x_1=0}^{1} \prod_{x_2=0}^{1} \sum_{x_3=0}^{1} \prod_{x_4=0}^{1} (x_1 + x_2 + x_3)(2 - x_1 - x_4)$$

This is just an integer (in fact 20). A proof would go like the following.

1. The prover chooses the prime 7 (in reality it should be larger, but we are only trying to illustrate the procedure). He claims that the expression is 6 modulo 7 and in fact that

$$\prod_{x_2=0}^{1} \sum_{x_3=0}^{1} \prod_{x_4=0}^{1} (x_1 + x_2 + x_3)(2 - x_1 - x_4)$$

as a function of x_1 is

$$P_1(x_1) = (2x_1^2 + 2x_1 + 1)(2x_1^2 + 6x_1 + 5)(1 - x_1)^2(2 - x_1)^2.$$

(Normally the prover represents these polynomials in a dense representation, but this is more convenient for hand calculation).

2. The verifier checks that $P_1(0) + P_1(1) \equiv 6$ modulo 7 (in the future we reduce everything modulo 7 without saying). Indeed $P_1(0) = 20 \equiv 6$ while $P_1(1) = 0$. He now chooses random value for x_1 (in our case $x_1 = 3$) and wants to be convinced that

$$\prod_{x_2=0}^{1} \sum_{x_3=0}^{1} \prod_{x_4=0}^{1} (3 + x_2 + x_3)(6 - x_4) = P(3) = 25 \times 41 \times 4 \times 1 \equiv 5.$$

3. The prover now claims that

$$\sum_{x_3=0}^{1} \prod_{x_4=0}^{1} (3 + x_2 + x_3)(6 - x_4)$$

as a function of x_2 is $P_2(x_2) = 1 + 4x_2^2$.

4. The verifier checks that $P_2(0)P_2(1) \equiv 5$ and randomly chooses $x_2 = 5$ and asks to be convinced that

$$\sum_{x_3=0}^{1} \prod_{x_4=0}^{1} (1 + x_3)(6 - x_4) \equiv P_2(5) \equiv 3$$

5. The prover claims that

$$\prod_{x_4=0}^{1} (1 + x_3)(6 - x_4)$$

as a a function of x_3 is $P_3(x_3) = 2 + 4x_3 + 2x_3^2$.

6. The verifier checks that $P_3(0) + P_3(1) \equiv 3$ and then randomly chooses $x_3 = 2$ and wants to be convinced that

$$\prod_{x_4=0}^{1} 3(6 - x_4) \equiv P_3(2) \equiv 4.$$

This he can to by himself and he accepts the input since 18×15 is indeed 4 modulo 7.

As mentioned before, the above proof does not relativize (the $IP \subset PSPACE$ does relativize, but not the second part). It is not difficult to construct an oracle A such that $IP^A \subset PSPACE^A$. The reason that the proof does not relativize is that if we allow oracle questions then the condition "C_1 is the configuration that follows C_2" cannot be described by a low degree polynomial.

This proof which does not relativize gives some hope to attack the NP vs P question. However it is still true that no strict inclusion that does not relativize has been proved for any complexity class that includes NC^1.