Advanced Algorithms CS3172, 02/03

Time: Monday and Friday, 10-11.

www: Do check the website regularly.

Lecture Course comes in 2 parts:
Part I (Dix) Introduction to Complexity Classes,
Part II (Rydeheard) Specific Algorithms.

Organisation:

Part I (Dix): 8 lectures, one week free (3/7 March), one week to discuss the homework (10/14 March).

Part II (Rydeheard): 8 lectures, one week free (5/9 May), one week to discuss the homework (12/16 May).

Exam:

Overview

1. Turing Machines
2. Complexity Classes
3. Hierarchies, Complete Problems

1 Turing Machines

1.1 The very definition

1.2 Computable languages

1.3 Modifications of TM's

1.4 Undecidability

1.1 The very definition

Computability: Consider functions over the natural numbers $\mathbb{N}$.

Which of these should we call computable?

Complexity: In order to measure the complexity of an algorithm, we need a machine model to check it against!

Which model do we choose?

Robustness: Model should not depend on small modifications.

It should be robust against such small changes.

One machine to rule them all!

This **ring of the rings** is the **Turing machine**, named after Alan Mathison Turing.

One can think of a Turing machine as a device, consisting of

Tape: A one-way infinite tape consisting of single cells (storage tape).

Head: A head which always scans exactly one cell. The head can move on the tape to the right or left and it can also print something on the cell that is currently scanned (thereby overwriting the contents of the cell).

Finite control: A mechanism that can be described by a finite table. It tells the head

1. to write a certain symbol on the current cell,
2. to move right or left and
3. to enter another state,

all this depending on the current state and the symbol scanned.

Definition 1.1 (Turing Machine, Version 1)

A deterministic Turing machine (DTM) is a tuple $\langle Q, \Sigma, \Gamma, \#, \delta, q_0 \rangle$:

- Q is a finite set (set of **states**).
- Γ (set of tape symbols) is a finite set with $\# \in \Gamma$,
- Σ (set of input symbols) is also a finite set with $\Sigma \subsetneq \Gamma$,
- $\#$ is a distinguished symbol (the blank) with $\# \notin \Sigma$.
- q_0 is a distinguished element of Q (start state): $q_0 \in Q$.
- δ is a function (the **next move function**) from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ which needs not to be defined on some inputs.

A DTM takes as input a word $w \in \Sigma^*$ and either halts after a number of moves (in which case the output is the word left on the tape) or it does not (in which case there is no output). The number of states of a DTM is $\|Q\| - 1$.

To be precise, we define the following:

ID: An **instantaneous description** of a DTM is denoted by the string $\alpha_1 q \alpha_2$ where $\alpha_1, \alpha_2 \in \Gamma^*$, $q \in Q$.

- Informally it means that the tape contains the string $\alpha_1\alpha_2$ (the first symbol of α_1 represents the first cell of the tape, and all cells beyond the last symbol of α_2 are blanks), the head is scanning the leftmost symbol of α_2 (if $\alpha_2 = \varepsilon$ then it is scanning a blank #) and the DTM is in state q.
- We also use a distinguished ID denoted by **stop:**$\alpha_1 q \alpha_2$: this means that $\alpha_1 q \alpha_2$ is the current description and the DTM stops.

An ID is like a **snapshot** of the current status of the DTM. It gives a complete description.

Move: For a DTM $\mathcal{M}$ we define a relation $\vdash_{\mathcal{M}}$ (a direct move) between ID's as follows. Firstly, stop:$\alpha_1 q \alpha_2 \vdash_{\mathcal{M}}$ stop:$\alpha_1 q \alpha_2$ for all $\alpha_1, \alpha_2 \in \Gamma^*$, $q \in Q$. Secondly, $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash_{\mathcal{M}}$

$$\vdash_{\mathcal{M}} \begin{cases} X_1 \cdots X_{i-1} Y p X_{i+1} \cdots X_n, & \text{if } i > 1,\ \delta(q, X_i) = (p, Y, R); \\ X_1 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n, & \text{if } i > 1,\ \delta(q, X_i) = (p, Y, L); \\ \text{stop:} q X_1 \cdots X_n, & \text{if } i = 1,\ \delta(q, X_i) = (p, Y, L); \\ X_1 \cdots X_{i-2} p X_{i-1} Y, & \text{if } i - 1 = n > 0,\ \delta(q, \#) = (p, Y, L); \\ \text{stop:} X_1 \cdots X_{i-1} q X_i \cdots X_n, & \text{if } \delta(q, X_i) \text{ is not defined ;} \end{cases}$$

We denote by $\vdash^*_{\mathcal{M}}$ the transitive closure of $\vdash_{\mathcal{M}}$. Thus for two ID's the relation

$ID_1 \vdash^*_{\mathcal{M}} ID_2$ means that ID_2 can be reached from ID_1 in finitely many steps.

Comp: Let a partial function $f : \Sigma^* \longrightarrow \Sigma^*;\ w \mapsto f(w)$ be given

(f needs not be defined on all inputs).

We say that a DTM $\mathcal{M}$ computes f,when the following holds:

$$f(w) = w' \text{ if and only if } \#wq_0 \vdash^*_{\mathcal{M}} \text{stop}:\#w'q,$$

where $q \in Q$.

Note that a DTM might not stop on certain inputs.

Definition 1.2 ((Partial) recursive functions over Σ)

A function computed by a DTM is called **partial recursive**. If such a function is defined on all inputs, it is called **recursive**.

The last definition was introduced by Alan Turing in his ground breaking paper (Turing 1936). **It is one of the most ingenious definitions in mathematics and logic of all times.**

Church's Thesis: The intuitive notion of an algorithm is correctly reflected by a Turing machine: The input is a word written on the tape and the output also is a word obtained after the machine stops.

The intuitive notion of a **computable function** over $\mathbb{N}$ is that of a **function computed by a Turing Machine** with $\Sigma = \{1\}$. Input and output are written in unary.

Visual Turing: `http://www.cheransoft.com/vturing/`

A nice tool to develop, play and understand Turing machines.

Note the differences between our definition and visual Turing:

1. variable assignments in visual Turing are represented in our definition by introducing additional states (one for each tape symbol: when an assignment α is made, we enter a particular state so that we know to remember α and can print it later on if needed).
2. In our model we must both write on the tape and move the head (in the same step).
3. In visual Turing subprograms can be easily incorporated, whereas in our definition we have to rename the states involved to avoid confusion (in particular the start state).

I should reach this point at the end of the first lecture. Finish with playing around with the visual Turing program.

Here is the definition of visual Turing's Left# wrt. $\Gamma = \{\#, a, b\}$

(Go to the first blank on the left of the current position of the head).

Let $Q := \{q_0, q_1\}$, $\Sigma := \{a, b\}$, and δ be defined as follows:

state	$\#$	a	b
q_0	$\langle q_1, \#, L\rangle$	$\langle q_1, a, L\rangle$	$\langle q_1, b, L\rangle$
q_1	$-$	$\langle q_1, a, L\rangle$	$\langle q_1, b, L\rangle$

- It is easy to design DTMs for addition, subtraction, multiplication, copying of strings etc. It might be tedious at first.
- DTMs can also simulate any type of subroutine, including recursive procedures and parameter passing mechanisms.
- I doubt whether you can come up with any number theoretic function that is not computable by a DTM. **Anything goes with a DTM.**

Homework 1 (Number of DTMs)

Given numbers n, m, exactly how many Turing machines with up to n states and m tape symbols are there?

How many Turing machines are there at all (no restriction on the number of states)?

Do the above answers change, if we allow the set Σ to be countably infinite?

How many instantaneous descriptions are there for a DTM working on a part of the tape of length n?

How many Turing machines or equivalents have actually been built from 1936-1986? I mean machines that can compute all partial recursive functions in the sense of Definition 1.2. Just make a guess.

Example 1.1 (Printing n on the tape: Print_n)

For each $n \in \mathbb{N}$, let us design a machine that prints exactly n 1's on the tape, starting with the empty tape standing on the second blank (from the left). We try to do it with as few states as possible. We use the machine copy. For given n, we first write $\lfloor \frac{n}{2} \rfloor$ many 1's on the tape. This can be done with $\lfloor \frac{n}{2} \rfloor$ many states. We then copy this number (this can be done with a constant number of states, namely 7). Then, we replace the separating # with a 1 and, depending on whether n is odd or even, we leave it as is or we replace the last 1 by a #. To do this we need another 2 states.

In total, we can write n 1s with only $\lfloor \frac{n}{2} \rfloor + 9$ states.

The machine Copy, using seven states:

state	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
#	$\langle q_1, \#, L\rangle$	$\langle q_2, \#, R\rangle$	$\langle q_7, \#, R\rangle$	$\langle q_4, \#, R\rangle$	$\langle q_5, 1, L\rangle$	$\langle q_6, \#, L\rangle$	$\langle q_2, 1, R\rangle$	$-$
1	$-$	$\langle q_1, 1, L\rangle$	$\langle q_3, \#, R\rangle$	$\langle q_3, 1, R\rangle$	$\langle q_4, 1, R\rangle$	$\langle q_5, 1, L\rangle$	$\langle q_6, 1, L\rangle$	$\langle q_7, 1, R\rangle$

1.2 Acceptable languages

In the last section we viewed a DTM as a computing device: given an input, it computes an output. Often, however, it is conceptually simpler to view a DTM as a device getting an input and returning *Yes* or *No*: this is called an **acceptor**.

A DTM under this viewpoint gets an input, runs and either stops in an accepting state, stops in an non-accepting state or runs forever. We therefore have to add a set of accepting states.

Definition 1.3 (Turing Machine, Version 2 (Acceptor))
A deterministic Turing machine (DTM) M is a seven-tuple

$$\langle Q, \Sigma, \Gamma, \#, \delta, q_0, F \rangle, \text{ where}$$

- Q is a finite set (set of states).
- Γ (set of tape symbols) is a finite set with $\# \in \Gamma$,
- Σ (set of input symbols) is also a finite set with $\Sigma \subsetneq \Gamma$,
- $\#$ is a distinguished symbol (the blank) with $\# \notin \Sigma$.
- q_0 is a distinguished element of Q (start state): $q_0 \in Q$.

- δ is a function (the next move function) from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ which needs not to be defined on some inputs.
- $F \subseteq Q$, F is the set of **final states**.

A DTM takes as input a word $w \in \Sigma^*$ and either (1) halts in an accepting state, or (2) halts in a non-accepting state, or (3) does not halt at all.

The only modification we have to make for the definition of a move (see slide 10) is the following:

If $q \in F$, then $X_1 \cdots X_{i-1} q X_i \cdots X_n \vdash_{\mathcal{M}}$ stop:$X_1 \cdots X_{i-1} q X_i \cdots X_n$.

Definition 1.4 (Language accepted by a DTM)
The language $\mathcal{L} \subseteq \Sigma^*$ accepted by a DTM M, denoted by $\mathcal{L}(M)$, is the set of words in Σ^* that cause M to enter a final state (when started on the string "#w" with the head at the first blank to the right of w).

Note that for non-accepted words, a DTM needs not halt!

Why are we talking about accepting languages? Because it is a nice unifying framework and makes the introduction of complexity notions so much easier!

Example 1.2 (Instances of Definition 1.4)

The following examples will be discussed and illustrated in greater depth later.

Numeric: $\Sigma_{\text{num}} := \{1\}$, $\mathcal{L}_{\text{primes}} := \{p : p \text{ is a prime}\}$. $\overline{\mathcal{L}_{\text{primes}}}$: the set of non-primes. Here we use unary notation.

One could also code integers in n-ary notation. We need an input alphabet $\Sigma := \{0, 1, \ldots, n-1\}$ and code an integer according to its value to the base n. For example, in 2-ary notation (binary), the integer 5 would be written as "101", whereas in unary notation it would be "11111".

Writing numbers in n-ary notation ($n > 1$) saves storage tape: instead of m cells we only need $\log_n m$ many.

However, there is only a gain when switching from unary to binary notation (from m to $\lg m$). Switching further to $\log_n m$ saves only a (linear) constant, but nothing more in the limit.

Formulae: $\Sigma_{sat} := \{\wedge, \vee, \neg, (,), x, 0, 1\}$. This is enough to define boolean expressions w over variables $\{x_1, x_2, \ldots\}$. A variable x_i is denoted be the string "x" followed by the binary notation of i written as 0s and 1's.

As usual, when the variables of such a boolean expression are set to **t** (true) or **f** (false), the whole expression evaluates to either true or false.

The satisfiability problem is:

Given an expression w, is it satisfiable?

Suppose we are given a boolean expression (in the usual form) with n symbols. What is the length of its coded version in Σ_{sat}?

There can only be $\lceil \frac{n}{2} \rceil$ different variables and each requires no more than $1 + \lceil \lg n \rceil$ symbols: this gives a bound of $n\lceil \lg n \rceil$. All our forthcoming complexity results will not depend on whether we are using n or $n \lg n$ as the length of the input (this will be explained later).

We then define:

$$\mathcal{L}_{sat} := \{w \in \Sigma^*_{sat} : \text{ there is an assignment of } x_i \text{ such that formula } w \text{ evaluates to true}\}$$

Graphs: Let $G = \langle V, E \rangle$ be a directed graph (V denotes the vertices and $E \subseteq V \times V$ the edges of the graph).

Is there a path leading from vertex 1 to vertex n?

We choose $\Sigma_{graph} := \{v, e, 0, 1, (,)\}$. v_i is represented as the string "v" followed by the binary notation of i written as 0s and 1's. An edge $e_{i,j}$ is represented as the string "(string1#string2)" where string1 stands for the binary representation of i and string2 stands for the binary representation of j. We define:

$$\mathcal{L}_{reach} := \{w \in \Sigma^*_{graph} : \text{ there is a path in graph } w \text{ leading from the first vertex } v_1 \text{ to the last one } v_n\}$$

Integer Linear Programming: Given an $m \times n$ matrix of integers $\mathbf{A}$ and a column vector $\mathbf{b}$.

Does there exist a column vector $\mathbf{x}$ such that $\mathbf{Ax} \geq \mathbf{b}$?

Representation is straightforward: words of the language are the entries of $\mathbf{A}$ and $\mathbf{b}$ written in binary.

$$\mathcal{L}_{\text{ILP}} := \{w \in \Sigma^*_{\text{ILP}} : \quad w \text{ represents an ILP problem } \langle \mathbf{A}, \mathbf{b} \rangle \text{ such that there is } \mathbf{x} \text{ with } \mathbf{Ax} \geq \mathbf{b}.\}$$

This is the point reached after the second lecture.

Is the language $\{ww^R : w \in \Sigma^*\}$ acceptable? Is the language $\emptyset$ acceptable?

What language does the following DTM accept:

state	#	a	b
q_0	$-$	$-$	$-$

What does the following DTM do:

state	#	1
q_0	$\langle q_1, 1, R\rangle$	$-$
q_1	$\langle q_2, 1, R\rangle$	$-$
q_2	$-$	$\langle q_1, a, L\rangle$

How many DTM's with 0 states are there?

Σ^*: Let Σ be finite or countable infinite. Then Σ^* is countably infinite. Each finite string can be represented as a natural number by using the prime factor decomposition.

Computable functions: There are as many computable functions as natural numbers.

$f : \mathbb{N} \to \{0,1\}$: There are uncountably many such functions. Given $f_1, \ldots f_n, \ldots$ we construct

$$C : \mathbb{N} \to \mathbb{N}; n \mapsto \begin{cases} 1, & \text{if } f_n(n) = 0; \\ 0, & \text{else.} \end{cases}$$

C is certainly different from all f_i (why?).

Subsets of $\mathbb{N}$: There are uncountably many subsets of $\mathbb{N}$.

Therefore there must be non-computable functions.

1.3 Modifications of DTM's

Various Modifications of DTMs have been defined. They are all equivalent when it comes to the class of computable functions or acceptable languages they lead to.

1.3.1 Modifying the tape

Two-way infinite tape: we can allow the tape to be two-way infinite. This adds even more storage and the head can not *fall off* when too far left.

Multitape DTM: We can allow multiple tapes (all of them two-way infinite). One can be distinguished as the input tape. In each step, all the heads on all tapes have to be moved (independently).

Theorem 1.1 (Two-way infinite = one-way infinite)

DTMs with a two-way infinite tape or with multiple such tapes are equivalent to ordinary DTMs: the set of acceptable languages is the same.

Proof:

two-way infinite: Obviously, a two-way infinite tape DTM can simulate an ordinary DTM: it just has to mark the left of its tape (the second blank to the left of the input) and can then simulate the programme of the DTM.

We have to show how to simulate a two-way infinite DTM on a DTM:

use all even numbered cells of the one-way tape to store the contents of the left hand side of the two-way tape, and all odd numbered cells for the right hand side.

We need some more states to remember on which tape we are and the simulation has to do two moves (to stay on the right tape) rather than just one. For each state q of the two-way infinite machine, we need two states $(q, even), (q, odd)$.

Multiple tapes: We have to show how to simulate a k-tape DTM on an ordinary one. As in the case of the two-way infinite tape, we split the tape into a number of tracks, namely $2k$ many.

Each tape of the k-tape DTM corresponds to 2 tracks: one just indicates the position of the head, and the other track corresponds to the contents of the tape.

It is quite obvious how this new DTM has to act to simulate the k-tape DTM (note that k is fixed: this information is stored in the states of the DTM).

Consider the language $\{ww^R : w \in \Sigma^*\}$ where w^R is the reverse of w.

1-tape DTM: How can this language be decided by a 1-tape DTM?
How many moves is the head doing?

2-tape DTM: How can this language be decided by a 2-tape DTM?
How many moves are the heads doing?

This technique shows that for m moves of the k-tape DTM, the 1-tape DTM needs a number of moves that is **quadratic in** m (actually, $6m^2$).

I should have reached this point after the third lecture.

1.3.2 Indeterminism

Our Definition 1.3 is deterministic in the sense that the move function δ is always uniquely determined (or undefined).

A nondeterministic version allows the machine to go into a finite number of successor states.

Definition 1.5 (Turing Machine, Nondeterministic Acceptor)
A **nondeterministic** Turing machine (NDTM) M is a seven-tuple $\langle Q, \Sigma, \Gamma, \#, \delta, q_0, F \rangle$ as in Definition 1.3, where δ is modified

- δ is a function (the next move function) from $Q \times \Sigma$ to $2^{(Q \times \Sigma \times \{L,R\})}$ which needs not to be defined on some inputs.

A NDTM takes as input a word $w \in \Sigma^*$ and either (1) halts in an accepting state, or (2) halts in a non-accepting state, or (3) does not halt at all.

The language $\mathcal{L} \subseteq \Sigma^*$ accepted by a NDTM M, denoted by $\mathcal{L}(M)$, is the set of words in Σ^* such that **there is a series of moves** that cause M to enter a final accepting state (when placed on the empty tape with the head at the first blank to the right).

Why are NDTM so important? The precise answer will be given in Section 3. But let us consider Example 1.2, the instance with the satisfiability problem.

Given a formula, to find whether it is satisfiable or not, we have to **guess** the right assignment of truth values for the variables.

We can model this very easily with a NDTM: whenever the NDTM reaches a variable, it makes two possible moves: one where the variable is set to true, the other where it is set to false.

Obviously, the formula is satisfiable if and only if there is a sequence of moves leading to an assignment making the whole formula true.

We want to construct a machine which checks whether the input wrt. $\Sigma = \{a, b\}$ is of the form ww, with $w \in \Sigma^*$.

The idea is to

1. find the middle position of the input, and
2. then successively replace the appropriate symbols (in the left word by #, in the right word by a new symbol c), and
3. checking whether they match.

We end up with a tape which consists of only cs (in which case the input had this form) or not (in which case the input was not of this form).

Here is the deterministic part of the program describing δ:

state	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
#	$\langle q_2, \#, R\rangle$		$\langle q_5, \#, R\rangle$	$\langle q_6, \#, R\rangle$				
a	$\langle q_2, a, R\rangle$	$\langle q_3, \mathbf{c}, L\rangle$	$\langle q_3, a, L\rangle$	$\langle q_4, a, L\rangle$	$\langle q_7, \#, R\rangle$		$\langle q_7, a, R\rangle$	$\langle q_1, a, L\rangle$
b	$\langle q_2, b, R\rangle$	$\langle q_4, \mathbf{c}, L\rangle$	$\langle q_3, b, L\rangle$	$\langle q_4, b, L\rangle$		$\langle q_7, \#, R\rangle$	$\langle q_7, b, R\rangle$	$\langle q_1, b, L\rangle$
c			$\langle q_3, c, L\rangle$	$\langle q_4, c, L\rangle$			$\langle q_8, c, R\rangle$	$\langle q_8, c, R\rangle$

How can you make this into a NDTM that correctly solves our problem?

Theorem 1.2 (DTM and NDTM are equivalent)

A language accepted by a NDTM is also accepted by some DTM.

Proof: Suppose we are given a NDTM. In order to construct an equivalent DTM, we have to make sure that all possible moves are simulated in a systematic manner. Note that there is a maximal number r of possible next-moves for our NDTM (determined by δ).

The simulation will be done on a 3-tape DTM. The first tape is for the input. The second tape systematically generates finite sequences of numbers from $1, \ldots, r$. The sequence $1, 5, 6, 3, 6$ means: in the simulation of the NDTM, when coming to the first choicepoint use possibility 1, for the second use 5, etc. for the fifth choicepoint use 6.

The third tape is for the actual simulation, where the choices are dictated by tape 2. ∎

1.3.3 Restrictions

Can we restrict the number of states or the number of symbols on the tape alphabet?

Definition 1.6 (Off-Line (N)DTM)
An off-line (N)DTM is a k-tape (N)DTM where one tape is the input tape. We assume the input is enclosed between two blanks #. The head of this tape can only move on the input and the enclosing blanks (to determine that the end of the input is reached), but not write or go farther to the left or right of the input.

Theorem 1.3 (Minimal number of tapes/Minimal alphabet)

Any acceptable language can be accepted by an off-line (N)DTM with one storage tape the alphabet of which is $\{\#, 1\}$.

Any acceptable language can be accepted by an off-line (N)DTM with one storage tape and 3 states.

1.4 Undecidability

As mentioned on slide 15, it is not easy to think of a function that is not computable, without sorting to very artificial examples. Here is one dating back to (Rado 1962; Lin and Rado 1965).

Define $BB : \mathbb{N} \to \mathbb{N}$ as follows

$$n \mapsto f(n) := \text{the maximal number of 1s a DTM (as in Definition 1.1 defined over } \Gamma = \{\#, 1\}) \text{ with maximal } n \text{ states can print on an empty tape and halt.}$$

It is easy to compute BB for $1, 2$: what results do you get?

The classical busy beaver is for two-way infinite tapes and final states (Definition 1.3). The only known values are (in our terminology): $BB(1) = 1, BB(2) = 4, BB(3) = 13$. Already $BB(5)$ is not known. And $BB(6)$ is greater than $1.29 * 10^{865}$.

Theorem 1.4 (BB is not computable)
The busy beaver increases too much to be computable: there is no DTM computing BB.

Proof: As one can always add 1 state to print one more 1:

- BB is strictly monotonically increasing.

Suppose there is a DTM BB computing BB. Let it have n_0 states.

We build a series of other machines which write $BB(m)$ 1s on the empty tape and use less than m states. This is a contradiction to our observation above.

Consider the compound machines $\mathrm{Comp_m} := \mathrm{BB} \circ \mathrm{Print_m}$, i.e. firstly m 1s are printed on the empty tape (see Example 1.1 on page 17) and then, secondly, BB runs to compute $BB(m)$. These machines have $n_0 + \lfloor \frac{m}{2} \rfloor + 9$ states. Now choose m_0 such that

$$m_0 > n_0 + \lfloor \frac{m_0}{2} \rfloor + 9.$$

Then for all $m \geq m_0$, Comp_m writes $BB(m)$ 1s on the tape and works with strictly less than m states. ∎

This is the point reached after the fourth lecture.

References

Hopcroft, J. and J. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison Wesely.

Lin, S. and T. Rado (1965). Computer studies of turing machine problems. *Journal of the ACM 12*(2), 196–212.

Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley.

Rado, T. (1962). On non-computable functions. *The Bell System Technical Journal XLI*(3), 877–884.

Turing, A. M. (1936). On Computable Numbers with an Application to the Entscheidungsproblem. *Proc. London Mathematical Soc., series 2 42*, 230–265. corrections ibid., 43:544–546.