Computational Complexity

Frank Stephan

Department of Computer Science

Department of Mathematics

National University of Singapore

fstephan@comp.nus.edu.sg

Overview

Lecture Dates in the Semester. The lectures are on Monday evenings followed by a tutorial (except Week 1)

13 Jan 2025	20 Jan 2025	27 Jan 2025	3 Feb 2025
10 Feb 2025	17 Feb 2025	3 Mar 2025	10 Mar 2025
17 Mar 2025	24 Mar 2025	7 Apr 2025	14 Apr 2025

Midterm Test: 17 March 2025, Second half of ninth lecture. 30 Marks.

Deadline for Homeworks to present: Last tutorial in Semester; however, it is strongly encouraged to do each week one homework.

Final Exam: 6 May 2025, 17.00-19.00 hrs, 60 Marks.

`www.comp.nus.edu.sg/~fstephan/complexity.html`

Homework and Task Rules

Marks: 1 marks per homework. Sum of all homework marks capped at 10 Marks. There are few tasks, these are comprehensive homeworks requiring more work and can be done by two students together, each sharing half of the load and each receives 1 marks for it.

In the Discussions Forum, students can write up only homeworks not done by others. Homeworks have in the thread title the following info (several students possible for tasks):‘

Homework Number, Student Name

For example:

Task 5.9 N. Immerman and R. Szelepcsényi

Check marks in Canvas in Reading Week.

Content 1-4

- Lecture 1 on Slide 7
  Machine models, Basic steps, Turing machines, Big and Little Oh, Counter machines, Addition machines of Floyd and Knuth.
- Lecture 2 on Slide 30
  Diagonalisation, Time hierarchies, Number of registers in Addition Machines, Computation Problems and Decision Problems.
- Lecture 3 on Slide 52
  Space Complexity Classes: LOGSPACE and its relationship to PTIME.
- Lecture 4 on Slide 77
  Nondeterminism and Alternation for logarithmic space. Savitch's Theorem. POLYLOGSPACE. Nick's Class.

Content 5-8

- Lecture 5 on Slide 107
  Deterministic and Nondeterministic Linear Space, Context-Sensitive Languages, the LBA problems and the Boolean closure of nondeterministic space.
- Lecture 6 on Slide 132
  P, NP and CoNP, in particular the world of NP-complete problems.
- Lecture 7 on Slide 156
  SAT-type Problems and Exponential Time Algorithms; The Exponential Time Hypothesis; Exponential Space Algorithms.
- Lecture 8 on Slide 182
  Counting Classes and Counting Algorithms; The Theorem of Valiant and Vazirani; Probabilistic Algorithms and the Class RP.

Content 9-12

- Lecture 9 on Slide 207
  Conditional lower bounds for polynomial time problems.
  Lecture 9 will be short and there will be the Midterm Test in the second half of the lecture time.
- Lecture 10 on Slide 223
  Lecture 10 will be on lower and upper bounds for deterministic and nondeterministic algorithms for problems in P.
- Lecture 11 on Slide 249
  Lecture 11 will be on the Polynomial Hierarchy, alternating polynomial time and the relations of these to PSPACE. A further topic is relativised worlds.
- Lecture 12 on Slide 275
  Lecture 12 will discuss the situation at PSPACE and beyond.

Lecture 1

Definition 1.1: Machine models are given by the machine commands plus the primitive steps which count as one step. The number of steps is then the runtime of the machine.

One basic model is that of a Turing machine. A Turing machine consists on one or several infinite tapes, a state, a finite set of tape symbols and commands. For the ease of notation, the machine is equipped with variables with constant range. In each step, it can either read the symbol of one of these tapes into a variable or scroll a tape in some direction or go (conditionally or unconditionally) to a line number (which might depend on the outcome of comparing a variable or constant with the symbol on the tape) or write a symbol or variable onto the tape.

Turing Machine Example 1.2

Example 1.2. This Turing machine only checks whether there are an even number of zeores and an even number of ones in the input word. It has bitvariables a, b and starts at the beginning of line 1.

```
Line 1: a=0; b=0;
Line 2: If tapesymbol is blank then
            begin Go forward; goto line 2 End;
Line 3: If tapesymbol is 0 then a = 1-a;
Line 4: If tapesymbol is 1 then b = 1-b;
Line 5: If tapesymbol is not blank then
            begin Go forward; goto line 3 end;
Line 6: If a=0 and b=0 then accept else reject.
```

The commands “accept” and “reject” communicate a decision of the Turing machine.

Turing Machine Example 1.3

```
Tape symbols 0,1,2,...,k and blank; one tape;
Output has to replace nonempty input word;
Variables a in {0,1,...,4*k},
  b,c in {0,1,...,k}; k > 5, k fixed.
Line 1: a=0; b=0; c=0; Goto 2;
Line 2: Go forward;
  if tapesymbol is blank then goto 2;
Line 3: Go forward;
  if tapesymbol is not blank then goto 3;
Line 4: Go backward once;
Line 5: Let b = max{0,tapesymbol}; %% blank<0
  a = a+b*3; c = a mod (k+1);
  a = a div (k+1); let tapesymbol = c;
Line 6: Go backward; If a > 0 or
  tapesymbol not blank then goto 5;
Line 7: Halt.
```

Notes 1.4

Technically, the state is a tuple holding all variable values, the current line number and the current position inside the line (if several instructions are in one line). The Turing machine program mandates does in each situation the following: It evaluates or updates the tapesymbol; it goes on the tape forward or backward (if explicitly said so); it updates variables; it goes to a new line number. Lines can have several commands, but a goto command always go to the beginning of a line. At the end of the line, the Turing machine goes to the next line.

One-tape Turing machines use the tape for everything: Finding the input written there; writing the output there and having necessary additional information on the tape during computation.

Example and Quiz

What does this Turing machine compute? It has the tapealphabet 0,1,2 and blank; two variables ranging over these four symbols and output has to replace nonempty input word. The “ts” is the tapesymbol.

```
Line 1: Go forward; if ts = blank then goto 1;
Line 2: Go forward; if ts != blank then goto 2;
Line 3: Go backward;
Line 4: Switch(ts) begin
  ts is 0: Let ts = 1; Goto Line 7;
  ts is 1: Let ts = 2; Goto Line 7;
  ts is 2: Let ts = 0; Goto Line 3;
  ts is blank: Let a=1; Goto Line 5 end;
Line 5: Go forward; Let b = ts;
  Let ts = a; Let a=b;
Line 6: If a is not blank then goto Line 5;
Line 7: Halt.
```

Ressource-Use of Turing Machine

Facts 1.5.
Size (Number of States) is bounded by product of:
(a) Overall number of instructions;
(b) Number of values for each constant ranged variable;
(c) (Tape alphabet size)$^{\text{Number of Tapes}}$.

Input size: Number of symbols except blanks on tapes where the input is.

Time used: Overall number of steps until halting.

Space used: Number of different fields visited on bidirectional tapes which can be modified.

If there are designated input and output tapes which are not used for computation: Do not count towards space use; the head on input tapes might be bidirectional, but the head on output tape goes only forward.

Time Usage Examples

Examples 1.6. Let $\mathbf{n}$ be length of input.

Multiplication with $\mathbf{3}$: Linear time, $\mathbf{O(n)}$.

Checking whether word is palindrome on one-tape machine: $\mathbf{O(n^2)}$.
Alternating forward pass and backward pass.
Forward pass: Find first unmarked input symbol and last unmarked input symbol and compare them, mark both after reading.
If symbols are equal then start backward pass else halt with output “not palindrome”.
Backward pass: Go backward until first unmarked symbol is found and go then to Forward pass.
In the case that forward pass or backward pass do not find unmarked symbols in the above process then halt with output “palindrome”.

Turing Machine Program

```
Line 1: Go forward; If tapesymbol blank
  then Goto 1;
Line 2: Read tapesymbol into a and mark as read;
Line 3: Go forward; if tapesymbol
  neither marked nor blank then goto 3;
Line 4: Go backward one step; if tapesymbol
  marked then goto Line 9;
Line 5: If tapesymbol different from a
  then goto 10;
Line 6: Mark tapesymbol as read;
Line 7: Go backward; If tapesymbol not marked
  then goto 7;
Line 8: Go forward; If tapesymbol not marked
  then goto 2;
Line 9: Accept (Is Palindrome); Halt;
Line 10: Reject (Is not Palindrome); Halt.
```

Upper Bound

Turing machine goes on word of length n at most n times forward and backward, as with each double-pass "forward-backward", two symbols get marked.

After $n/2+1$ double-passes, all symbols are marked and the algorithm has reached a decision, either palindrome or not palindrome.

If it says "not palindrome" after k forward passes, then the k-th symbols from front and from end are different, thus the output is correct – here note that the k-th forward pass marks the k-th symbols from front and from end after inspecting them.

If it says "palindrome" then no k has been found such that the k-th symbol from front and from end are different, thus the output is also correct.

Lower Bound

Suppose a one-tape Turing machine wants to to recognise the set $\{u0^n u : |u| = n\}$. How to prove that it needs at least quadratic time?

The method is that of a crossing sequence. Given i, j with $i + j = n$, the crossing sequence is the sequence of all states when the Turing machine crosses the border between $v0^i$ and $0^j w$ for words v, w of length n in order to check whether $v = w$.

Assume now that the crossing sequences for $v0^n v$ and $w0^n w$ are the same for some i, j with $i + j = n$. Then the machine produces on $v0^n v$ and $v0^n w$ the same output. Thus there must be at least 2^n different crossing sequences and each crossing sequence has at least linear length. Due to $n + 1$ choices for (i, j), the Turing machine has to cross a border between v and w $\Omega(n^2)$ times.

The O-Calculus (Landau Notation)

Definitions 1.7. Assume a Turing machine uses, in the worst case, $\mathbf{T(n)}$ steps for inputs of length $\mathbf{n}$. Then $\mathbf{T(n) \in O(f(n))}$ iff there is a constant $\mathbf{c}$ with $\mathbf{T(n) \leq c \cdot f(n)}$ for all $\mathbf{n \geq c}$. Furthermore, $\mathbf{T(n) \in \Omega(f(n))}$ iff there is a constant $\mathbf{c > 1}$ with $\mathbf{T(n) > f(n)/c}$ for all $\mathbf{n \geq c}$. $\mathbf{T(n) \in o(f(n))}$ iff for all $\mathbf{c > 1}$ there is a constant $\mathbf{d}$ such that for all $\mathbf{n \geq d}$, $\mathbf{T(n) < f(n)/c}$. $\mathbf{T(n)} \in \omega(\mathbf{f(n)})$ iff for all $\mathbf{c > 1}$ there is a constant $\mathbf{d}$ such that for all $\mathbf{n \geq d}$, $\mathbf{T(n) > f(n) \cdot c}$.

Alternative definition by Hardy and Littlewood: $\mathbf{T(n) \in \Omega(f(n))}$ iff there is a constant $\mathbf{c > 1}$ with $\mathbf{T(n) \geq f(n)/c}$ for infinitely many $\mathbf{n \geq c}$ and $\mathbf{T(n)} \in \omega(\mathbf{f(n)})$ iff for every constant $\mathbf{c > 1}$ there are infinitely many $\mathbf{n}$ with $\mathbf{T(n) > f(n) \cdot c}$. These versions are referred to as $\mathbf{\Omega'}$ and ω' from now onwards.

$\mathbf{T \in \Theta(f(n))}$ iff $\mathbf{T(n) \in O(f(n))}$ and $\mathbf{T(n) \in \Omega(f(n))}$.

Sample Rules: If $\mathbf{T, T' \in O(f(n))}$ and $\mathbf{T'' \in O(g(n))}$ and $\mathbf{c > 0}$ then $\mathbf{T(n) + T'(n) \in O(f(n))}$, $\mathbf{c \cdot T(n) \in O(f(n))}$ and $\mathbf{T(n) \cdot T''(n) \in O(f(n) \cdot g(n))}$.
$\mathbf{T(n) \notin O(f(n))} \Leftrightarrow \mathbf{T(n)} \in \omega'(\mathbf{f(n)})$.

Examples for O-Calculus

Examples 1.8. Polynomials: $\mathbf{n^k \in O(n^h)}$ iff $\mathbf{k \leq h}$.
$\mathbf{O(p((k+1)n)) = O(p(n))}$. $\mathbf{n^4 \in \Theta(5n^4 + 1881)}$. $\mathbf{n^7 \notin O(n^5)}$.
$\mathbf{O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in \Theta(g(n))}$.
$\mathbf{1551 \notin O(1/n)}$. $\mathbf{n + n^2 + n^3 \in O(n^4)}$. $\mathbf{n^4 \notin O(n)}$.

Examples 1.9. Exponentials and Factorials:
$\mathbf{2^{n+1} \in O(2^n)}$, $\mathbf{2^{n+\log n} \notin O(2^n)}$, $\mathbf{3^n \in \Omega(2^n)}$, $\mathbf{3^n \notin O(2^n)}$.
For all $\mathbf{k > 1}$: $\mathbf{n^n \notin O(k^n)}$, $\mathbf{n^n \in \Omega(k^n)}$, $\mathbf{(n+1)^{n+1} \notin O(n^n)}$.
$\mathbf{n! \in O(n^n)}$, $\mathbf{(n/4)^n \in O(n!)}$, $\mathbf{(n+1)! \notin O(n!)}$.
For Euler Number $\mathbf{e = 2.71828\ldots}$, $\mathbf{n! \in \Theta((n/e)^n)}$.

Examples 1.10. Step Functions:
$\mathbf{T(0) = 1}$, $\mathbf{T(2n+1) = T(2n)}$, $\mathbf{T(2n+2) = 2^{T(2n+1)}}$,
$\mathbf{f(n) = T(n+1)}$, $\mathbf{g(0) = 1}$, $\mathbf{g(n+1) = T(n)}$:
$\mathbf{T(n) \in O(f(n))}$, $\mathbf{T(n) \in \Omega'(f(n))}$, $\mathbf{T(n) \notin \Omega(f(n))}$.
$\mathbf{T(n) \notin O(g(n))}$, $\mathbf{T(n) \in \Omega(g(n))}$, $\mathbf{T(n) \in \omega'(g(n))}$.

Counter and Addition Machines

Definition 1.11: Counter machines can increment (add 1), decrement (minus 1) and compare with 0 (outcomes positive, zero, negative) their counters. Besides that they are written with programs like Turing machines and can also have additional constant-range variables which one can compile away into a longer (and less understandable) program.

Counter machines are Turing complete from the viewpoint of recursion theory, however, they need exponential time for adding two n-digit integers.

Definitions 1.12: Addition machines are counter machines which can in addition add and subtract and which can compare either variables or variables and constants (outcome greater, equal, smaller). For those, polynomial time coincides with polynomial time of Turing machines.

Counter Machine Examples 1.13

The following counter machine reads the inputs x, y and outputs their sum z. As computing the sum overwrites the values of x and y, it keeps for book keeping purposes the copy v of x and w of y.

```
Line 1: Read x; Read y;
Line 2: Let z=0; Let v=0; Let w=0;
Line 3: If x = 0 then goto Line 4;
   Decr x; Incr z; Incr v; Goto Line 3;
Line 4: If y = 0 then goto Line 5;
   Decr y; Incr z; Incr w; Goto Line 4;
Line 5: Write z;
```

Using these techniques, one can indeed implement every addition machine as a counter machine with sufficiently many counters.

Convention 1.14 of Addition Machines

In order to simplify the writing of addition machines, the following is allowed:

- Constant ranged variables use letters at the beginning of the alphabet and integer ranged registers use letters at the end of the alphabet; the possible values of variables must be listed;
- Integer constants and constant ranged variables can be used for comparing, subtracting, adding, assignment;
- Assignments can have any number of terms on the right side, furthermore, one can write $\mathbf{x = 3y - 2z}$ in place of $\mathbf{x = y + y + y - z - z}$;
- One can use “begin” and “end” in if-then-else statements;
- All goto statements go to the beginning of some line.

Counting Trailing Zeroes

Note that one has no direct access to the digits of the input x, one has to compute them. This construction reads out the digits of x at the top and counts the trailing zeroes.

```
Line 1: Read x; If x < 1 Then Goto 1;
        Let y=1; Let z=10; Let w=0;
Line 2: Let y = 10y; If y < x+1 Then Goto 2;
Line 3: Let x = 10x; Let z=10z;
Line 4: If x = 0 then Goto 5; If x < y Then
        goto 3; Let x = x-y; Goto 4;
Line 5: If z = y Then Goto 6;
        Let w = w+1; Let z = 10z; Goto 5;
Line 6: Write w; Halt.
```

Here x has the input, y is the first power of 10 larger than x. One uses y to remove the digits of x which are shifted out on the top. z is a counter which counts by multiplying with 10; w is, at the end, the result.

Multiply with Addition Machines 1.15

```
Example: Multiply 1234 with 111, Running Sum
                                 with Frontshifts
  111                            0 + 111 =    111
   222                        1110 + 222 =   1332
    333                      13320 + 333 =  13653
     444                    136530 + 444 = 136974
  ------
  136974
```

Algorithm: Read digits of one number (here 1234) out at the top and add product of digit with other number to running sum times 10.

Use binary numbers to avoid multiplication by digits 2,3,4,...,9 and decide only on adding the second number (bit 1) or skipping the addition of the second number (bit 0).

Example with Coding Digit

Example of Multiplication:

```
12340 (x) times 111 (y) Digit Running sum
1000000 (Comparator v)   -              w=0
1234010                  1              111
2340100                  2             1332
3401000                  3            13653
4010000                  4           136974
0100000                  0          1369740
1000000   Coding digit reached, the End
```

Algorithm x times y:

1. Append Coding digit 1 to x;
2. Comparator v is power of 10 greater x;
3. Shift x up, while ($x > v$) do x=x-v;
4. Digit is number of subtractions needed;
5. Shift w up, Add digit times y to w;
6. If x = v then stop else goto 3.

Program for Multiplication

1. Begin Read $\mathbf{x}$; Read $\mathbf{y}$; Let $\mathbf{a = 1}$;
2. If $\mathbf{x < 0}$ Then Begin Let $\mathbf{x = -x}$; Let $\mathbf{a = -a}$ End;
3. If $\mathbf{y < 0}$ Then Begin Let $\mathbf{y = -y}$; Let $\mathbf{a = -a}$ End;
4. If $\mathbf{y < x}$ Then Begin Let $\mathbf{v = x}$; Let $\mathbf{x = y}$; Let $\mathbf{y = v}$ End;
5. Let $\mathbf{v = 1}$; Let $\mathbf{w = 0}$; Let $\mathbf{x = x + x}$; Let $\mathbf{x = x + 1}$;
6. If $\mathbf{v > x}$ Then Goto $\mathbf{7}$;
   Let $\mathbf{v = v + v}$; Goto $\mathbf{6}$;
7. Let $\mathbf{x = x + x}$; If $\mathbf{v = x}$ Then Goto $\mathbf{8}$ Else Let $\mathbf{w = w + w}$;
   If $\mathbf{x > v}$ Then Begin Let $\mathbf{w = w + y}$; Let $\mathbf{x = x - v}$ End;
   Goto $\mathbf{7}$;
8. If $\mathbf{a = -1}$ Then Begin Let $\mathbf{w = -w}$ End;
   Write $\mathbf{w}$; End.

Homeworks

Homework 1.16: Provide a two-tape Turing machine which checks whether the input (on the first tape) is a palindrome in linear time (using accept / reject to communicate result).

Homework 1.17: Prove a quadratic lower bound for palindrome-checking with one-tape Turing machines.

Homework 1.18: Provide a quadratic time one-tape Turing machine which recognises whether the input is of the form $\mathbf{3u2u2u2u2u4}$ where $\mathbf{u}$ is a binary string and the tape alphabet is decimals plus blank.

Homework 1.19: Provide a quadratic time one-tape Turing machines computing the square of the binary number on the input.

Homework 1.20: Provide a proof that one cannot improve this algorithm to subquadratic time for squaring on one-tape Turing machines.

Homeworks

Homework 1.21: Provide a read-only one-tape Turing machine which accepts an input word if and only if it has three 1s and five 2s and arbitrarily many 0s and no other digits.

Homework 1.22: Assume that $\mathbf{f(n) > 0}$ for all $\mathbf{n}$. Prove that $\mathbf{T(n) \in O(f(n))}$ iff there is a constant $\mathbf{c}$ such that, for all $\mathbf{n}$, $\mathbf{T(n) \leq c \cdot f(n)}$. Explain why the assumption that the domain is the set of natural numbers is needed.

Homework 1.23: Provide a counter machine which computes the square of an input number. Provide a lower bound on the computation time needed (in terms of the number $\mathbf{n}$ of binary digits needed to write down the input number).

Homeworks

Homework 1.24: Provide an addition machine which checks whether at least one of five input numbers to be read satisfies that its ternary representation uses all three ternary digits $0, 1, 2$ after a leading digit which is either 1 or 2. Try to keep the number of registers needed minimal.

Homework 1.25: Write a linear time addition machine which checks whether the input x into a product of two numbers y and z with y being a power of 2 and w being a power of 3. If this is the case, the machine should accept and output v and w. Otherwise the machine should reject.

Homework 1.26: Prove identities in Examples 1.8.

Homework 1.27: Prove identities in Examples 1.9.

Homework 1.28: Prove identities in Examples 1.10.

Tasks

Task 1.29: Provide a 2-counter machine which checks whether the input number x given is a palindrome when written as a ternary number. Give the program explicitly, but the usage of constant ranged variables is allowed. Compute a bound on the state complexity of the counter machine as follows: Count for each line number the number of instructions in the line and sum these up; multiply the resulting number with the number of possible values of each constant-ranged variable used. Explain the construction in detail.

Lecture 2

This lecture is about Complexity Aspects of Addition Machines. Diagonalisation, Time and Space hierarchies, Computation Problems and Decision Problems.

The central topic in this chapter is to compare complexity classes. Most complexity classes are defined in dependence of certain parameters:

First: What machine model is used;

Second: Which aspects of the machine is during the complexity of the computation is evaluated for the time used or the space used or, in the case of addition machines, the number of registered needed to carry out the computation fast.

Third: Size of input - usually number of symbols of word or binary digits of number.

Complexity of Turing Machines

Time: The maximum number of steps carried out by the Turing machine on an input of n symbols.

Space: The overall number of cells visited on tapes which can be modified and where the head can move in both directions.

Tapes: The number of tapes needed to perform a task under certain side-conditions. All things can be computed on a one-tape Turing machine, if no further requirements are given, but, for example, recognising palindromes in linear time requires two tapes. There is a whole hierarchy of problems which requires so and so many tapes for computations which satisfy certain side conditions.

Complexity of Counter Machines

Time: In the normal range, uninteresting, as even adding two numbers requires exponential time. However, for larger classes with preassigned space constraints (like polynomial space or linear space), one might try to measure the time used up by a counter machine.

Space: The space used by a counter machine is the number of binary digits of the largest register value which appeared during the computation process. This is a reasonable but uncommon measure.

Number of Registers ("Counters"): This has been investigated thoroughly. If there are no further constraints, two counters are enough.

Complexity of Addition Machines

Time: How much time does the machine need to computer a function, say $f(x, y)$, from x and y? Input size is measured as number of binary digits needed to write out the whole input (with blanks separating several input numbers). Speed is measured in number of additions, subtractions and comparisons and other commands carried out.

Space: Size of the largest register during running time in dependence of the input size.

Number of Registers: Registers must hold inputs and outputs and for one-input functions, two registers are enough without any side-conditions – see Counter Machines. Floyd and Knuth [1990] investigated the number of registers needed for linear time addition machines. Value depends on the exact constraints which operations are allowed and which not.

Lower Bound for Halving

Theorem 2.1 [Stockmeyer 1976]. An addition machine needs at least linear time for the mapping $x \mapsto x/2$ where one downrounds the result in the case that x is odd.

Proof. One considers numbers modulo x. The size of a number y modulo x is the minimum of the number of binary digits which are needed to write y and $x - y$, where $0 \leq y < x$. So one starts with input x and follows a given algorithm. In each instruction, a new value of a register might be computed which is the sum or difference of old values. The number of digits (modulo x) increases at most by a 1 in each addition or subtraction. Thus one needs $\log(x/2) - O(1)$ basic operations (adding and subtracting), where $O(1)$ comes from integer constants which form the start of the induction.

Stockmeyer [1976] obtained the same lower bound for checking whether the input is even or odd.

Upper Bound

This algorithm divides by two. A small adjustment would then also detect whether the number is even or odd (last value of a). The algorithm is for positive numbers and zero.

```
Line 1: Read x; Let x = x+x+1; Let z=0;
  Let y = 1; Let b=0 (a,b are bitvariables);
Line 2: Let y = y+y; If y <= x then goto 2;
Line 3: Let x = x+x; If x = y then goto 4;
  If y < x then begin Let x = x-y; let a=1; end
    else let a=0;
  Let z = z+z; Let z = z+b; let b = a; Goto 3;
Line 4: Write z.
```

For input x, one appends a coding bit and reads out the bits of x at the top using y. The bits will be copied into z except for the last bit read, that one will be discarded. The variables a, b are there to store the bits intermediately.

Other Linear Time Operations

Register Complexity: How many registers does one need for linear time operations?

Multiplication: 4 registers sufficient;
Division: 4 registers sufficient;
Remainder: 3 registers sufficient; in the model of Floyd and Knuth (usage of constant 1 counted as one additional register, at most one operation + or - per assignment, only comparisons of two registers) also three registers needed;
Computing Fixed Remainders (Simulation of Finite Automata): 2 registers;
Outputting powers of 2 in binary representation: 4 registers sufficient.

Open Problem: Is there a constant number of registers sufficient for all linear time operations? Most likely not.

Constant Time Operations

Constant time operations carry out only constantly many steps on an addition machine. They are functions without loops. Example of sorting three numbers.

```
Line 1: Read x; Read y; Read z;
Line 2: If x > y then
  begin let u = x; let x = y; let y = u end;
Line 3: If y > z then
  begin let u = x; let x = y; let y = u end;
Line 4: If x > y then
  begin let u = x; let x = y; let y = u end;
Line 5: Write x; Write y; Write z.
```

Four registers suffice to sort three inputs in constant time.

Constant Time Operations

One can do the same even with three registers. Algorithm:

```
Read x, y, z;
Choose the unique case which applies:
If x <= y and y <= z then Write x,y,z;
If x <= z and z <  y then Write x,z,y;
If y <  x and x <= z then Write y,x,z;
If y <= z and z <  x then write y,z,x;
If z <  x and x <= y then write z,x,y;
If z <  y and y <  x then write z,y,x.
```

One uses “less or equal” when the order of register names coincides with their value order and “strictly less” otherwise to avoid double listings.

Hierarchy

Theorem 2.2. One can read in and sort and write out n variables in constant time if and only if one has at least n registers. Thus the function mapping n inputs to n outputs which sorts these n numbers in constant time witnesses that one needs at least n registers.

Proof. The sufficiency is obtained by generalising the previous example from 3 to n inputs; note that the inputs are read in some order and might be output in the inverse order, thus one needs to memorise all n inputs and one cannot make any memory manipulation operations (like storing two numbers in one register) in constant time. So for constant time, there is a strict hierarchy of what can be computed and memorised in this time.

Addition Machine Simulation

Theorem 2.3. There is a constant $\mathbf{k}$ such that, given a number $\mathbf{m}$ of registers, then there is a universal machine with $\mathbf{m}+\mathbf{k}$ registers which on inputs a number $\mathbf{e}$ and then simulates the $\mathbf{e}$-th program in some enumeration of all register machine programs such that, whenever the original program uses on inputs of size $\mathbf{n}$ the time $\mathbf{f(n)}$ then the simulating program uses time $\mathbf{O(f(n)\cdot size(e))}$ to do the same computation. In particular if $\mathbf{f(n)}$ is bounded by a constant $\mathbf{c}$ for all inputs then the simulating machine just needs $\mathbf{O(size(e))}$ to run the program, independently of the input size.

Proof idea I

One uses in addition to the given $\mathbf{m}$ registers $\mathbf{k}$ additional registers which have the same function as accumulators in central processing units and then one reads from the program (which can be a machine translation of a human readable program) what to do, for example which registers to copy into an accumulator or vice versa and what operations to do with the accumulators, for example subtract or add the accumulator $\mathbf{B}$ to $\mathbf{A}$ and to store the result in accumulator $\mathbf{C}$ or to compare the two accumulators. It is also part of the program to code integer constants (say in binary) in order to use them as operands or as line numbers in conditional or unconditional Goto commands.

Proof Idea II

As the program does not modify the register content except for loading the registers into accumulators and then do the corresponding operation (addition or subtraction) with them in one step and as constants are bounded by the size of e, the overall simulation of a step is just bounded by a function proportional in $\mathrm{size}(e)$ and not influenced by the register content in the simulated machine.

2.4 Simulating Addition Machines I

Task 2.4: Assume that the following commands are allowed for a register machine, where $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are registers and $\mathbf{y}, \mathbf{z}$ can also be integer constants: $\mathbf{x} = \mathbf{y} + \mathbf{z}$, $\mathbf{x} = \mathbf{y} - \mathbf{z}$, unconditional goto, if-then-else commands in dependence of conditions $\mathbf{y} < \mathbf{z}$, $\mathbf{y} = \mathbf{z}$, $\mathbf{y} > \mathbf{z}$, reading and writing of a register. Construct a register program in some suitable programming language - for example, Python allows arbitrarily large registers - and use additional constant range variables to reduce the program size so that the simulation bounds from the preceding theorem are met. State the size of the constant $\mathbf{k}$, that is, the number of additional registers, explicitly, constant size variables do not count towards this value $\mathbf{k}$.

2.4 Simulating Addition Machines II

Task 2.4 (continued): For the writing of the program, any machine coding of the program can be used, as long as it allows to simulate any step handling registers and accumulators and integer parameters in $\mathbf{O(size\ t)}$ steps of the simulating machine where the parameter $\mathbf{t}$ is the maximum of all integer constants involved and the value of register and line number labels; furthermore, a conditional or unconditional goto command might also invoke operations of size $\mathbf{O(size\ e)}$ due to the scrolling of the program (which is held in constantly many registers read out at the top in some fixed digital coding).

In the case that Python is used, the program could be augmented by a translator which translates a program, given as text, into the number $\mathbf{e}$. For other programming languages, use string data type and explain how to simulate it.

Addition Machine Diagonalisation

Given an input of length $\mathbf{n}$, one can compute two numbers $\mathbf{d}$, $\mathbf{e}$ used for diagonalisation as follows:

```
Line 1: Read x; Let y=1; Let e=1; Let d=1;
Line 2: If d>e then let e=e+1 else
          begin d=d+1; e=1 end;
Line 3: Let y=y+y; If y < x then Goto 2;
```

Using these parameters, one makes a program which does the following:

```
Line 4: Simulate the Universal Machine for
  (log d) times n steps on input x using
  h lines; for this use double up counting
  in the same way as for getting n out of x;
  ....
Line 5+h: If the simulation stops with result y
  then Write y+1 else Write 0.
```

Formal Statement

Theorem 2.5. There is an addition machine simulating the $\mathbf{e}$-th $\mathbf{O(n)}$ $\mathbf{k}$-register register machine for $\mathbf{n \cdot \log d}$ steps on infinitely many $\mathbf{x}$ and outputting on these $\mathbf{x}$ something different. The computation time of the inputs for fixed $\mathbf{d, e}$ is $\mathbf{n \cdot \log d \cdot O(size\ e)}$ and is inside $\mathbf{O(n \log^2 n)}$ in general.

Corollary 2.6. Some addition machine with $\mathbf{k} + \ell$ registers can compute in time $\mathbf{O(n \log^2 n)}$ a function different functions computed on $\mathbf{k}$ register addition machines in time $\mathbf{O(n)}$.

Compressing several registers into one for storing purposes uses up linear time and similarly for retrieving or modifying the value of one of the stored registers. Thus the number of registers is an essential ingredient for above result.

Low Turing Machine Complexities

Theorem One-tape Turing machines either need $\mathbf{O(n)}$ steps or $\mathbf{\Omega'(n\ log\ n)}$ steps.

The proof method analyses crossing sequences. Let $\mathbf{x_m}$ be the length-lexicographically first word with a crossing sequence of length $\mathbf{m}$. If one cuts $\mathbf{x_m}$ into three parts $\mathbf{u, v, w}$ such that the longest crossing sequence occurs inside $\mathbf{u}$ and at the borders from $\mathbf{u}$ to $\mathbf{v}$ and $\mathbf{v}$ to $\mathbf{w}$ occur the same crossing seuqences, then the word $\mathbf{uw}$ still has the same longest crossing sequence as $\mathbf{uvw}$ and thus $\mathbf{uvw}$ is not $\mathbf{x_m}$. So in $\mathbf{x_m}$ there are at least $\mathbf{|x_m|/2 - 1/2}$ different crossing sequences. If $\mathbf{c}$ is the number of states of the Turing machine and $\mathbf{(c+1)^k < |x_n|/2 - 1/2}$ then half or more of the crossing sequences have length $\mathbf{k}$. Let $\mathbf{n = |x + m|}$; now $\mathbf{k \geq log\ n\ /\ log(c+1) - 1}$. The $\mathbf{x_m}$ witness that the runtime is at least $\mathbf{\Omega'(n\ log\ n)}$.

Computation and Decision Problems

A computation problem or functional computation problem is the computational complexity to compute a certain function. For example, Floyd and Knuth showed that in the addition machine model for linear time computations, the number of registers needed is an important complexity measure (besides the time constraint). The side-condition "linear time" is important, as otherwise many problems like multiplication and remainder have trivial algorithms.

Computation Problem: The problem to compute a certain function, often lower bounds are obtained by analysing output size.

Decision Problem: The problem to compute a $\{0, 1\}$-valued function, here one cannot drive up the complexity by big outputs. Such functions can be viewed as characteristic functions of sets or relations.

Homeworks 2.7-2.8

Homework 2.7: Explain how to simulate the following operations with a register machine: Holding a string in $\mathbf{k}$-ary alphabet as $\mathbf{k}$-ary numbers in two registers with a third register larger than these and one symbol in a constant-sized variable and how to use this string as a program which is used to read out binary numbers as well as other symbols and how to scroll it until a certain lable is found.

Homework 2.8: Construct a timer which reads in one input $\mathbf{x}$ of size $\mathbf{n}$ which is the first number with $\mathbf{2^n > x}$ and then counts some time variables for $\mathbf{n^3}$ steps until the counting comes to an end. Between each two steps it carries out some routine (omitted and to be indicated by dots) and it aborts the program with output $\mathbf{0}$ if it halts and the routing returns output $\mathbf{y+1}$ if it computes output $\mathbf{y}$ within the given time.

Homeworks 2.9-2.13

Construct Addition Machines which compute as fast as possible the following functions, where the input $\mathbf{x} \geq \mathbf{1}$. Homeworks 2.9 and 2.10 run linear in the size of the output.

Homework 2.9: $\mathbf{x} \mapsto \mathbf{x}^{\mathbf{x}}$.

Homework 2.10: $\mathbf{x} \mapsto \mathbf{x}!$ where $\mathbf{x}!$ is the product of all natural numbers from $\mathbf{1}$ to $\mathbf{x}$.

Homework 2.11: $\mathbf{x} \mapsto \mathbf{loglog}(\mathbf{x})$ where the logarithm is the first number $\mathbf{y}$ with $\mathbf{2}^{\mathbf{y}} \geq \mathbf{x}$.

Homework 2.12: $\mathbf{x} \mapsto \mathbf{y}$ where $\mathbf{x}$ and $\mathbf{y}$ have each nonzero digit the same number of times in their decimal representations and the digits of $\mathbf{y}$ are ordered ascendingly. So $\mathbf{203318}$ is mapped to $\mathbf{12338}$. Explain the algorithm.

Homework 2.13: The same as 2.12, but with ternary numbers and give the explicit program.

Homeworks 2.14-2.17

Homework 2.14: Construct a Turing machine which uses one tape and simulates a two-tape Turing machine. Show if the two-tape Turing machine uses $\mathbf{O(f(n))}$ time than the one-tape Turing machine uses $\mathbf{O((n+f(n))\cdot f(n))}$ time. Provide a reason for the delay.

Homework 2.15: Can an exponential time counter machine simulate an exponential time Addition machine where all register stay linear in size of the input during the whole computation? Explain the result.

Homework 2.16: Provide a function which can be computed by an exponential time addition machine but not by an exponential time counter machine.

Homework 2.17: Provide asymptotically fast programs to compute $\mathbf{x \mapsto \log(x)}$ with $\mathbf{\log(x) = \min\{y : 2^y \geq x\}}$ on counter machine, addition machines and Turing machines.

Lecture 3

This lecture is about space complexity. Usually it is measured by the space used on a Turing machine with the following items:

1. One Bidirectional Input Tape;
2. One Bidirectional Worktape of which only $\mathbf{bound(n)}$ symbols can be used where $\mathbf{n}$ is the length of the word on the input tape (the Turing machine has to take track of this);
3. One Unidirectional Output Tape. Whenever its head writes a symbol, it afterwards automatically moves forward one step and it cannot move otherwise, initially the full tape was blank.

The Turing machine might have two states "Accept" and "Reject" for answer $\mathbf{1}$ and answer $\mathbf{0}$ instead of an output tape and it halts upon going into one of these states.

Space Complexity Classes

The bound function $\mathbf{bound}$ can be from a certain class $\mathbf{BOUND}$ of permitted bound functions and one can use such classes to define space complexity classes.

LOGSPACE: $\mathbf{BOUND}$ is the class of all functions $\mathbf{f}$ for which there is a constant $\mathbf{c}$ with $\mathbf{f(n) \leq c \log(n)}$ for all $\mathbf{n \geq 2}$.

LINSPACE: $\mathbf{BOUND}$ is the class of all functions $\mathbf{f}$ for which there is a constant $\mathbf{c}$ with $\mathbf{f(n) \leq c\, n}$ for all $\mathbf{n \geq 2}$.

PSPACE: $\mathbf{BOUND}$ is the class of all polynomials in one variable $\mathbf{n}$.

ESPACE and EXPSPACE: $\mathbf{BOUND}$ is the class of functions $\mathbf{2^{f(n)}}$ where $\mathbf{f}$ is either a linear function or a polynomial, respectively.

Addition machines can only be used for complexity classes from LINSPACE onwards.

Refined Complexity Bounds

For Turing machine complexity, one takes mainly into account the following three complexity parameters:

1. Number of Worktapes (has some influence);
2. Number of Cells visited on work tapes (space measure);
3. Number of computation steps done (time complexity).

For Addition machines, the complexity measures are the following:

1. Number of registers;
2. Maximum size of a register during computation (space measure);
3. Number of computation steps done (time complexity).

If one limits the space then the computation time can become worse, as the next example shows.

Palindrome LOGSPACE Example

Theorem 3.1. Assume that a Turing machine has one bi-directional input tape (read only), $\mathbf{k}$ LOGSPACE work tapes and special Accept / Reject commands (or states). Then the time complexity of recognising palindromes on such a machine is $\mathbf{\Theta(n^2/\log(n))}$.

Algorithm. The algorithm holds the following numbers on the work tapes: Number $\mathbf{n}$ of cells of the input word (one forward and backward scanning of for counting all); the number $\mathbf{j = \log(n)}$ of digits to store $\mathbf{n}$; the number $\mathbf{i \cdot j}$ of digits from front and back already compared; the current $\mathbf{j}$ digits to be compared.

Each of these numbers can be stored on an own work tape, but one could also unify all these numbers with some logarithmic overhead onto one work tape to maintain them.

Algorithm Continued

One initialises $\mathbf{n}$ and its logarithm $\mathbf{j}$ by counting the digits of the input in a forward-backward scan.

Then, for $\mathbf{i = 0, 1, \ldots, n/j - 2}$, one reads digits at positions $\mathbf{i \cdot j, i \cdot j + 1, \ldots, i \cdot j + j - 1}$ and stores them on the corresponding work tape. Then one goes to positions $\mathbf{n - i \cdot j, n - i \cdot j - 1, \ldots, n - i \cdot j - j + 1}$ and compares the digits there with those on the work tape.

If all comparisons show coincidence then the word is a palindrome and one says $\mathbf{ACCEPT}$ else it is not a palindrome and one says $\mathbf{REJECT}$.

Time Analysis. Note that the input tape is only scrolled (from beginning to end and back) $\mathbf{O(n/\log n)}$ times and that each such process of scrolling and comparing takes $\mathbf{O(n)}$ time; thus the overall effort is $\mathbf{O(n^2/\log n)}$.

Lower Bound I

For this, one uses that there are only $\mathbf{c \log n}$ symbols on the work tape. Now one incorporates the work tape content into the Turing machine states and obtains a one-tape read-only Turing machine with $\mathbf{n^{c'}}$ states for some sufficiently large $\mathbf{c'}$ independent of $\mathbf{n}$. Now one considers words $\mathbf{v0^nw}$ where $\mathbf{v, w}$ are binary words. The word $\mathbf{v0^nw}$ has to be accepted iff $\mathbf{w}$ is the mirror image of $\mathbf{v}$. If there is now an $\mathbf{m \in \{0, 1, \ldots, n\}}$ where the crossing sequence length $\mathbf{n'}$ satisfies for $\mathbf{2^n/(n+1)}$ words $\mathbf{v0^nv^{mi}}$ that $\mathbf{(n^{c'})^{n'} < 2^n/(n+1)}$ then there are two such words $\mathbf{v0^nv^{mi}}$ and $\mathbf{w0^nw^{mi}}$ sharing the same crossing sequence and either one of these inputs is rejected or the input $\mathbf{v0^nw^{mi}}$ is accepted, both is a mistake.

Lower Bound II

Thus one has for all $\mathbf{m}$ and for all but less than $\mathbf{2^n/(n+1)}$ words $\mathbf{v0^nv^{mi}}$ that the crossing sequence has a length $\mathbf{n'}$ such that $\mathbf{n' \cdot c' \cdot \log n \geq n - \log n}$. This condition implies $\mathbf{n' \geq n/(2c' \log n)}$. In particular there is one word $\mathbf{v0^nv^{mi}}$ satisfying this for all $\mathbf{m}$. This word witnesses that on an input of length $\mathbf{3n}$ the algorithm runs at least $\mathbf{\Omega(n^2/\log n)}$ steps.

Checking Multiplications I

Definition 3.2: The problem MULTIPLICATION CHECK: Given a product of two numbers and its supposed result, check whether the result is correct.

Example: $12345 * 67890 = 838102050$.

Theorem 3.3: MULTIPLICATION CHECK is in LOGSPACE.

Note that one does not need to do the full computation for checking. It is sufficient to check modulo numbers $m = 2, 3, \ldots, n^2$ where n is five plus the number of binary digits. Accepts iff all modulo tests show equality.

Note that LOGSPACE is enough to hold variables containing n (input length plus five), any value $m \leq n^2$ and i, j, k being the remainders by m of the two factors and the supposed product, respectively.

Checking Multiplications II

The remainder $\mathbf{i}$ of $\mathbf{x}$ by $\mathbf{m}$ for binary $\mathbf{x}$ is computed as follows:

```
Line 1: Let s = 0; Let i=0;
Line 2: Read digit d of x; Let i=i+i+d;
Line 3: If i+1>m Then Let i=i-m;
Line 4: If x not completely read go to Line 2;
```

The equation whether $\mathbf{i \cdot j = k}$ modulo $\mathbf{m}$ can be checked as follows:

```
Line 1: Let s = 0; Let h=0;
Line 2: Let s = s+j; Let h=h+1;
Line 3: If s < m Then Goto Line 4 Else s=s-m;
Line 4: If h < i Then Goto Line 2;
Line 5: Now i*j=k Modulo m iff s=k.
```

All variables involved are in $\mathbf{\{0, 1, \ldots, 2m\}}$ and can be stored in LOGSPACE.

Prime Number Condition

Theorem 3.4: If $m_1, m_2, \ldots, m_n$ do not have a common factor and are all at least 2 then two numbers x and y below the product of $m_1, m_2, \ldots, m_n$ are equal if and only if they are equal modulo each of $m_1, m_2, \ldots, m_n$.

Theorem 3.4 is known as the "Chinese Remainder Theorem".

Fact 3.5: There are at least n prime numbers below n^2 for $n \geq 5$. The product of these n primes is at least 2^n, as each prime is at least 2.

Theorem 3.4 together with Fact 3.5 are sufficient to prove that the algorithm above is correct, note that all binary numbers have at most n digits. Thus MULTIPLICATION CHECK is in LOGSPACE.

Classes in Logarithmic Space

The following classes are also in LOGSPACE (without proof).

USTCON (Omer Reingold 2008): Given an undirected graph (V, E) and two nodes $s, t \in V$, is there a path from s to t?

REPETITION: Given a list of binary numbers separated by commas, are two of these numbers equal?

CLAUSE SET EVALUATION: List of clauses using $x_1, x_2, \ldots, x_n$ plus list of Boolean values of these variables. Check whether all clauses are satisfied.

BRACKET EXPRESSION CORRECTNESS: Given an expressions using brackets, are all the bracket placed in a matching order? One can have even different types of brackets.

Alternative Machine Models

The following machine models are equivalent to LOGSPACE, that is, recognise exactly the languages in LOGSPACE.

1. One-Tape Read-Only Turing Machines with Multiple Reading Heads. The heads can detect whether they are on the same field and whether they have reached the beginning or the end of the Input.

2. One-Tape Read-Only Turing Machine with additional Addition Machine Registers whose values are always bounded by the input length raised to the power of some constant. So this is some type of hybrid machine. The hybrid machine has explict Accept and Reject commands in its programming language and uses only the symbols on the Input Tape as input. Registers can be added, subtracted, assigned and compared; operations with constants and tape symbols are possible.

Constant Space

Constant Space Machines have only one Read-Only Turing tape for input and explicit Accept and Reject commands. It can be required that the input is read in one direction only. The languages which can be recognised in CONSTANT SPACE are exactly the regular languages.

Regular languages can be characterised by grammars satisfying severe restrictions or by Finite Automata (which are Read-Only One-Way Turing machines) and regular expressions.

Regular expressions are either finite set of explicitly listed words or obtained from other regular expressions by one of the following constructs: Union, Intersection, Concatenation, Kleene Star, Set Difference. Here Kleene Star $\mathbf{L}^*$ of a language $\mathbf{L}$ is the set of all words obtained by concatenating some words from $\mathbf{L}$.

Configurations of Turing machines

Definition 3.6: A configuration of a Turing machine is given as follows:

1. The state (line number and constant-ranged variables) of the Turing machine;
2. The position on the input tape;
3. The content of the work tapes.

One can now compute that for an $\mathbf{f(n)}$ space machine there are constants $\mathbf{c', c}$ such that the machine has at most $\mathbf{(n+1) \cdot c' \cdot c^{f(n)}}$ configurations. If one configuration appears twice in a computation, the computation does not terminate. Thus the $\mathbf{f(n)}$ space Turing machine either runs forever or terminates after at most $\mathbf{(n+1) \cdot c' \cdot c^{f(n)}}$ steps.

Thus LOGSPACE is contained in P (=PTIME) and PSPACE is contained in EXP (=EXPTIME).

LOGSPACE versus P

Open Problem 3.7: Is P = LOGSPACE? This problem is open since the 1970ies and is not yet resolved.

However, researchers have identified many P-complete problems. These are problems with the following property: If A and B are problems in P and B is P-complete, then there is a LOGSPACE computable function f with $x \in A$ if and only if $f(x) \in B$.

It is conjectured (though not proven) that no P-complete problem is in LOGSPACE or some other small space class.

The next slide provides the formal definitions for two P-complete problems, the CIRCUIT VALUE PROBLEM and the MONOTONE CIRCUIT VALUE PROBLEM.

The Circuit Value Problems

Definition 3.8: A circuit is a list $(\mathbf{B_0}, \mathbf{B_1}, \ldots, \mathbf{B_n})$ of gates together with Boolean equations of the following type for each $\mathbf{m = 0, 1, \ldots, n}$: $\mathbf{B_m = 0}$, $\mathbf{B_m = 1}$, $\mathbf{B_m = \neg B_k}$ for some $\mathbf{k < m}$, $\mathbf{B_m = B_i \vee B_j}$ for some $\mathbf{i, j < m}$, $\mathbf{B_m = B_i \wedge B_j}$ for some $\mathbf{i, j < m}$; for each $\mathbf{m}$ exactly one equation is given. The value of $\mathbf{B_n}$ (according to the equations) is called the value of the circuit.

A circuit without equations of the type $\mathbf{B_m = \neg B_k}$ is called monotone.

The CIRCUIT VALUE PROBLEM: Given a circuit, determine its value.

The MONOTONE CIRCUIT VALUE PROBLEM: Given a monotone circuit, determine its value.

Context-Free Grammars I

A context-free grammar consists of a terminal alphabet Σ, a nonterminal alphabet Δ, a start symbol $S \in \Sigma$ and rules $A \rightarrow w$ where $A \in \Delta$ and w is a word of some symbols from Σ and Δ. The language generated by a context-free language is the language of all words over the alphabet Σ which can be obtained by starting with S and by replacing some member $A \in \Delta$ in the current word which also occurs in a rule $A \rightarrow w$ by the word w until all digits of the current word are in Σ.

Example: $\Sigma = \{0, 1, 2\}$, $\Delta = \{S\}$, the rules are $S \rightarrow 0S1$, $S \rightarrow 22$. Now the words $22, 0221, 002211, 00022111$ and so on are in the language L generated by this grammar. For example, $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \rightarrow 00022111$ is generated by iteratively applying the above rules.

Context-Free Grammars II

Further Examples: Language of all ternary words with as many $\mathbf{0}$ as $\mathbf{1}$, but arbitrarily many $\mathbf{2}$.
$\mathbf{\Sigma = \{0, 1, 2\}}$, $\mathbf{\Delta = \{S\}}$, the rules are $\mathbf{S \to SS|S0S1S|S1S0S|2|\varepsilon}$. One can write $\mathbf{S \to v|w}$ in place of $\mathbf{S \to v, S \to w}$ to denote two or more alternatives.

All ternary words of either the form $\mathbf{0^n 1^n}$ or $\mathbf{0^n 2^n}$ with $\mathbf{n > 2}$.
$\mathbf{\Sigma = \{0, 1, 2\}}$, $\mathbf{\Delta = \{S, T, U\}}$, the rules are $\mathbf{S \to T|U}$, $\mathbf{T \to 0T1|000111}$, $\mathbf{U \to 0U2|000222}$. Sample derivations: $\mathbf{S \Rightarrow T \Rightarrow 000111}$, $\mathbf{S \Rightarrow U \Rightarrow 0U2 \Rightarrow 00002222}$.

One writes $\mathbf{v \Rightarrow^* w}$ to indicate that $\mathbf{w}$ can be derived from $\mathbf{v}$ in arbitrarily many derivation steps, where the case $\mathbf{v = w}$ is included and means in $\mathbf{0}$ derivation steps. The language of a grammar $\mathbf{G}$ is the set of all words $\mathbf{w \in \Sigma^*}$ which satisfy $\mathbf{S \Rightarrow^* w}$.

Example

Theorem 3.9: The following problem is P-complete (UNIFORM CONTEXT-FREE GRAMMAR MEMBERSHIP): Given a word $\mathbf{w} \in \mathbf{\Sigma}^*$ and a context-free grammar $(\mathbf{\Sigma}, \mathbf{\Delta}, \mathbf{S}, \mathbf{R})$ – here $\mathbf{R}$ is the set of rules –, is the word $\mathbf{w}$ generated by this grammar?

Proof by reducing the MONOTONE CIRCUIT VALUE PROBLEM to this problem.

Given the circuit $(\mathbf{B_0}, \mathbf{B_1}, \ldots, \mathbf{B_n})$ and its rules, one creates grammar $(\mathbf{\Sigma}, \{\mathbf{B_0}, \mathbf{B_1}, \ldots, \mathbf{B_n}\}, \mathbf{B_n}, \mathbf{R})$ with the below rules:

Rule in Circuit	Rules in Grammar
$\mathbf{B_m = 0}$	none
$\mathbf{B_m = 1}$	$\mathbf{B_m} \rightarrow \varepsilon$
$\mathbf{B_m = B_i \vee B_j}$	$\mathbf{B_m \rightarrow B_i, B_m \rightarrow B_j}$
$\mathbf{B_m = B_i \wedge B_j}$	$\mathbf{B_m \rightarrow B_i B_j}$

Does this grammar generate the empty word ε?

Simple Game Winning Strategy

A SIMPLE GAME is a game where one can move, from a node n, downwards with one or two choices in each move. The choices will be on two players, Anke and Boris. As the game always goes down, there are at most n moves. Nodes without outgoing edges evaluate either to “Anke” or to “Boris” to decide the game.

SIMPLE GAME aka SIMPLE GAME WINNING STRATEGY: Which player has a chance to win a simple game?

Here a winning strategy is a function which, based on passed choices of the players, advises the player on the current choices so that the player wins.

Theorem 3.10: (a) The SIMPLE GAME has a polynomial time algorithm; (b) There is a reduction of the MONOTONE CIRCUIT VALUE PROBLEM to SIMPLE GAME WINNING STRATEGY.

Solving the Simple Game

One marks from level 0 up to level n which player has a winning strategy. If nodes $0, 1, \ldots, m-1$ are marked with either Anke or Boris, then one does on node m the following:

If Anke chooses where to go and one successors of m is marked as an “Anke Node”, then m is an “Anke Node”.

If Anke chooses where to go and all successors of m are marked as “Boris Node”, then m is a “Boris Node”.

If Boris chooses where to go and one successors of m is marked as a “Boris Node”, then m is a “Boris Node”.

If Boris chooses where to go and all successors of m are marked as “Anke Node”, then m is an “Anke Node”.

Once the node n is marked, its name gives the player with the winning strategy who always moves to own nodes.

P-Completeness

One reduces instances $(\mathbf{B_0}, \mathbf{B_1}, \ldots, \mathbf{B_n})$ of the MONOTONE CIRCUIT VALUE PROBLEM to the SIMPLE GAME by translating the circuit rule update for a gate $\mathbf{B_m}$ into the action for node $\mathbf{m}$ as follows:

Rule in Circuit	Action in Game
$\mathbf{B_m = 0}$	Anke wins
$\mathbf{B_m = 1}$	Boris wins
$\mathbf{B_m = B_i \vee B_j}$	Boris chooses node $\mathbf{i}$ or node $\mathbf{j}$
$\mathbf{B_m = B_i \wedge B_j}$	Anke chooses node $\mathbf{i}$ or node $\mathbf{j}$

The game starts in node $\mathbf{n}$. Which player has a winning strategy?

Homeworks 3.11-13

For these homeworks, recall that multi-head Turing machines know when two heads stand on the same cell and one might assume special commands to place one head at the beginning of the input or the position of another head. Constant-ranged variables are allowed.

Homework 3.11: Write a Turing program for a four-head read-only Turing machine recognising REPETITION.

Homework 3.12: Write a Turing program for a four-head read-only Turing machine recognising BRACKET EXPRESSION CORRECTNESS with symbol $\mathbf{a}$ (not a bracket) and three types of brackets: $\{, \}, (,), [,]$.

Homework 3.13: Fix a syntax for clause sets and write a Turing machine program for CLAUSE SET EVALUATION; use as few heads as possible.

Homeworks 3.14-3.16

For these homeworks, use hybrid Turing-Addition Machines with one input tape two-sided scrollable and as few additon machine registers as possible.

Homework 3.14: Write a Turing program for a hybrid machine recognising REPETITION.

Homework 3.15: Write a Turing program for a hybrid machine recognising BRACKET EXPRESSION CORRECTNESS with symbol $\mathbf{a}$ (not a bracket) and three types of brackets: $\{,\}, (,), [,]$.

Homework 3.16: Fix a syntax for clause sets and write a Turing program for a hybrid machine for CLAUSE SET EVALUATION.

Homeworks 3.17-3.19

Homework 3.17: Write a Turing program for a hybrid machine interpreting a LOGSPACE two-tape Turing machine; the work tape must be simulated by addition machine registers. Choose a suitable format for the program and data on the input tape.

Homework 3.18: Write a Turing program for a multi-head machine interpreting a LOGSPACE two-tape Turing machine; explain the construction and the usage of the heads; assume that the worktape is binary and not longer than $\log n$ symbols. Choose a suitable format for the program and data on the input tape.

Homework 3.19: Explain how to simulate a $k \cdot \log n$ space LOGSPACE machine with a binary worktape by a multi-head machine. How does the number of heads depend on k?

Lecture 4

Nondeterminism and Alternation. In the history of computing and mathematics, the ideas of nondeterminism and alternation are quite important. Economic theories deal a lot with two-player games; these are, to a certain degree, just another framework of alternation. Furthermore, one-player games are also a mirror image of nondetermism. Some instance of a problem can be solved nondeterministically if the solution can be found by some player where the rules of the game say which moves are allowed to obtain a solution.

Whether nondeterminism helps with polynomial time computations is the central question of the P-NP problem; here solutions can be checked in polynomial time, but finding them requires a winning strategy for the corresponding game which might not be in P.

Alternating Logarithmic Space

Definition 4.1: ALTERNATING LOGSPACE or ALOGSPACE [Chandra, Kozen and Stockmeyer 1981] is the complexity class of problems solvable by LOGSPACE computations in which two players can, at certain positions, choose how the computation should go on. More precisely, the Turing machine has Anke-Choice and Boris-Choice GOTO commands. A word $\mathbf{x}$ on the input tape is than in the given language $\mathbf{L}$ if Boris can do his choices such that the outcome is "ACCEPT", irrespective of what Anke does; furthermore, $\mathbf{x} \notin \mathbf{L}$ if Anke can do her choices such that the outcome is "REJECT", irrespective of what Boris does.

Similarly one can define ALTERNATING POLYNOMIAL TIME (AP) and ALTERNATING POLYNOMIAL SPACE (APSPACE) and other such complexity classes.

Repetition: Simple Game

A SIMPLE GAME is a game where one can move, from a node n, downwards with one or two choices in each move. The choices will be on two players, Anke and Boris. As the game always goes down, there are at most n moves. Nodes without outgoing edges evaluate either to “Anke” or to “Boris” to decide the game.

SIMPLE GAME aka SIMPLE GAME WINNING STRATEGY: Which player has a chance to win a simple game?

Here a winning strategy is a function which, based on passed choices of the players, advises the player on the current choices so that the player wins.

Theorem 3.10: (a) The SIMPLE GAME has a polynomial time algorithm; (b) There is a reduction of the MONOTONE CIRCUIT VALUE PROBLEM to SIMPLE GAME WINNING STRATEGY.

Alternating Logspace

Theorem 4.2: SIMPLE GAME is ALOGSPACE complete.

Proof: (a) SIMPLE GAME is in ALOGSPACE. An instance of SIMPLE GAME could be represented as a computer program of numbered lines with four types of commands as in this example:

1. Anke Choice Goto 2,5;
2. Boris Choice Goto 3,7;
3. Anke Choice Goto 4,6;
4. Anke Choice Goto 5,6;
5. Boris Choice Goto 6,7;
6. Accept and Halt (Boris wins);
7. Reject and Halt (Anke wins).

Simple Game Code

One can use single letters for the commands: A ("Anke Choice Goto"), B ("Boris Choice Goto"), H ("Accept and Halt"), R ("Reject and Halt"). The input text of the game on the previous slide is

1A2,5; 2B3,7; 3A4,6; 4A5,6; 5B6,7; 6H; 7R.

where the spaces are optional. Decimal numbers need LOGSPACE to write down and GOTOs go only in one direction. A alternating Turing machine would Halt and Accept iff player Boris has a winning strategy enabling him to reach a line with the command "Accept and Halt".

Task 4.3

Task 4.3: Write a Turing machine program interpretating SIMPLE GAME programs as in the short form

1A2,5; 2B3,7; 3A4,6; 4A5,6; 5B6,7; 6H; 7R.

on a two-tape Turing machine with the input in the first tape having the input and the second tape the label of the current instruction. For commands “A” and “B”, the players Anke and Boris have to provide the input whether the first or the second number should be replacing the current one in the work tape. This simulation interacts with players (let them choose the way). Once the program is done, explain how it proves that SIMPLE GAME is in ALTERNATING LOGSPACE in some few sentences.

Alternating Space Completeness I

Assume that a Turing machine with one input tape and k work tapes witnesses that the language L is in ALTERNATING LOGSPACE. Now one considers the set of configurations of the Turing machine, as defined in Definition 3.2. As the content of each work tape is bounded by $k' \cdot \log n$ for some constant k' where n is the length of the input, there are only $n^{k''}$ different configurations for some constant k''. Thus if player Boris can enforce that an alternating Turing reaches an accepting state then it can do it within $2n^{k''}$ steps, as the computation should not repeat a configuration for the same player, as otherwise either player Boris can enforce a faster way to acception or player Anke can enforce an infinite loop.

Alternating Space Completeness II

Thus one can consider all pairs $(\mathbf{i},\mathbf{j})$ of configurations where each transition goes from some state $(\mathbf{i},\mathbf{j})$ to $(\mathbf{i}+\mathbf{1},\mathbf{j}')$ with either $\mathbf{j}'$ being the successor configuration of $\mathbf{j}$ – if there are several possible $\mathbf{j}'$ then the configuration $\mathbf{j}$ player specified which player can choose accordingly – or $\mathbf{j}'=\mathbf{j}$ in the case that the game that the simulated Turing machine has reached one of “Accept and Halt” or “Reject and Halt”.

When defining $(\mathbf{i},\mathbf{j}) = \mathbf{i}\cdot\mathbf{2n^{k''}}+\mathbf{j}$ then the above game has the configuration numbers always going up and thus does not go in loops. Furthermore, the start state is $(\mathbf{0},\mathbf{m})$ for the starting configuration $\mathbf{m}$ of the simulated machine and $(\mathbf{i},\mathbf{j})$ is a halting state iff $\mathbf{j}$ is one for the simulated machine and $\mathbf{i}=\mathbf{2n^{k''}}$; a halting configuration $(\mathbf{i},\mathbf{j})$ is accepting iff $\mathbf{j}$ is; the Turing machine abstains in non-halting configurations of the form $(\mathbf{2n^{k''}},\mathbf{j})$. (End of Proof of Theorem 4.2)

Nondeterministic Logspace

An important special case of alternating computations are nondeterministic computations where only one player is used. NONDETERMINISTIC LOGSPACE, in short NLOGSPACE, is the complexity class of all languages L for which there is a two-tape Turing machine with a LOGSPACE workspace which uses only one player, say Boris, such that a word x is in L if and only if this player finds a way to guide the computation by choices until it accepts the computation.

Clearly all LOGSPACE problems are in NLOGSPACE, since on can just make Turing machines which officially use a player but never ask him for advice; similarly NLOGSPACE is in ALOGSPACE what equals P.

NLOGSPACE Complete Problems

The P-complete problems from before are in NLOGSPACE if and only if NLOGSPACE = P. Similarly there are NLOGSPACE complete problems which are in LOGSPACE if and only if LOGSPACE = NLOGSPACE.

The most famous NLOGSPACE complete problem are STCON and the bounded version of USTCON.

Definition 4.4: Given a directed graph (V, E) and nodes $s, t \in V$, STCON is the problem to determine whether there is a path from s to t in the graph.

Definition 4.5: Given an undirected graph (V, E), nodes s, t and a number m, is it possible to reach the node t from s within m steps? This problem is called BUSTCON or bounded s-t connectivity in an undirected graph.

Membership in Logarithmic Space

Consider an instance $(\mathbf{V}, \mathbf{E}, \mathbf{s}, \mathbf{t}, \mathbf{m})$ of BSTCON (bounded $\mathbf{s}$-$\mathbf{t}$ connectivity) which just drops the condition “undirected” from BUSTCON. Then the following NLOGSPACE algorithm solves this problem:

```
Line 1: Let v=s (node); Let k=0 (counter);
Line 2: If v=t then Halt and Accept;
Line 3: Let Boris select a successor w of v;
Line 4: Let k=k+1; Let v=w;
Line 5: If k<m+1 Then Goto Line 2;
Line 6: Halt without accepting.
```

This NLOGSPACE computation accepts the input $(\mathbf{V}, \mathbf{E}, \mathbf{s}, \mathbf{t}, \mathbf{m})$ if and only if Boris can find a way of going from $\mathbf{s}$ to $\mathbf{t}$ within $\mathbf{m}$ steps. If such a path exists, Boris can help to find it and the computation accepts; if such a path does not exist, Boris cannot give an advice to do the impossible.

Completeness of BUSTCON I

Proposition 4.6: STCON is LOGSPACE reducible to BUSTCON, that is, a LOGSPACE Turing machine can compute an equivalent BUSTCON instance from a STCON instance.

Constructing the Graph
Given a graph $(\mathbf{V}, \mathbf{E})$ where $\mathbf{V}$ has $\mathbf{n}$ elements, one makes the following graph $(\mathbf{W}, \mathbf{F})$ where $\mathbf{W} = \mathbf{V} \cdot \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{n}\}$ and $\mathbf{F}$ contains all pairs $((\mathbf{x}, \mathbf{k}), (\mathbf{y}, \mathbf{k}+\mathbf{1})), ((\mathbf{y}, \mathbf{k}+\mathbf{1}), (\mathbf{x}, \mathbf{k}))$ of nodes in $\mathbf{W}$ where either $\mathbf{x} = \mathbf{y}$ or there is an edge from $\mathbf{x}$ to $\mathbf{y}$ in the directed graph $(\mathbf{V}, \mathbf{E})$. Note that the graph $(\mathbf{W}, \mathbf{F})$ is undirected by the way it is defined (each edge in $\mathbf{W}$ has also its mirror image in $\mathbf{W}$, thus allowing the transition in both ways). The idea is that going in $(\mathbf{W}, \mathbf{F})$ against the direction of $(\mathbf{V}, \mathbf{E})$ forces to go down in the second coordinate so that too much time is lost.

Completeness of BUSTCON II

Now if one can go from $(\mathbf{s}, \mathbf{0})$ to $(\mathbf{t}, \mathbf{m})$ in $(\mathbf{W}, \mathbf{F})$ within $\mathbf{m}$ steps, then the second coordinate must in each step be counted up by one. Thus, except for transitions from $(\mathbf{x}, \mathbf{k})$ to $(\mathbf{x}, \mathbf{k}+\mathbf{1})$, each transition is of the form $(\mathbf{x}, \mathbf{k})$ to $(\mathbf{y}, \mathbf{k}+\mathbf{1})$ for some $(\mathbf{x}, \mathbf{y}) \in \mathbf{E}$, thus the path in the graph $(\mathbf{W}, \mathbf{F})$ is the image of a directed path in $(\mathbf{V}, \mathbf{E})$ of length up to $\mathbf{m}$; the not needed steps can be used to rest on some $\mathbf{x}$ while counting up the second coordinate.

Note that the second coordinate does not need to be larger than $\mathbf{n}$ as every reachable node in $(\mathbf{V}, \mathbf{E})$ can be reached from $\mathbf{s}$ in at most $\mathbf{n}$ steps by skipping all loops. Thus $\mathbf{t}$ is reachable from $\mathbf{s}$ in $(\mathbf{V}, \mathbf{E})$ iff $(\mathbf{t}, \mathbf{n})$ is reachable from $(\mathbf{s}, \mathbf{0})$ in $(\mathbf{W}, \mathbf{F})$ within $\mathbf{n}$ steps.

Savitch's Theorem

Walter Savitch (1970) found the following theorem:

Theorem 4.7: NLOGSPACE can be solved in deterministic space $(\log^2(n))$ and the corresponding simulation takes time $n^{O(\log(n))}$.

The important part of this result is the space usage, without that constraint, NLOGSPACE is a subset of P. Again one sees, as with palindrome solving, that it might happen that an algorithm saves computation space at the expense of computation time.

The algorithm is first illustrated for a STCON solver.

Example for Savitch's Theorem

The following algorithm solves STCON. Assume $\mathbf{n = 2^k}$.

```
Start Algorithm with call Connect(s,t,n).
Line 1: Recursive Algo Connect(s',t',n')
Line 2: If s'=t' or (0<n' and (s',t') in E)
  Then Return True;
Line 3: If n'>0 Then n''=n'/2 Else Return False;
Line 4: For all u' in V Do Begin
  If Connect(s',u',n'') and Connect(u',t',n'')
  Then Return True End;
Line 5: Line 4 failed for all u'; Return False.
```

Depth of Calls is $\mathbf{k = \log(n)}$. Each call stores local variables $\mathbf{s', t', n', n'', u'}$. These need $\mathbf{5\log(n)}$ space. Overall space $\mathbf{O(\log^2(n))}$. Timebound per call: $\mathbf{O(n)}$ times subcalls. Thus $\mathbf{O(n)^k = n^{O(\log(n))}}$.

Proof of Savitch's Theorem

Assume an NLOGSPACE algorithm is given with space bound $\mathbf{c \cdot \log(n)}$ on the tape. Then there are at most $\mathbf{n^d}$ configurations for some constant $\mathbf{d}$. This is also a bound on the nondeterministic computation time.

Now let $\mathbf{V}$ be the set of all configurations and $\mathbf{E}$ be the set of pairs of configurations $\mathbf{(x, y)}$ such that either the Turing machine can go from one configuration $\mathbf{x}$ to the other one $\mathbf{y}$ in at most one step. This graph can be computed in LOGSPACE, instead of outputting the graph, one can access for each pair $\mathbf{(x, y)}$ of configuration the reduction and check whether $\mathbf{(x, y)}$ is in the set $\mathbf{E}$.

The previous' slide algorithm solves STCON for $\mathbf{(V, E)}$ in space $\mathbf{O(\log^2(n^d)) = O(\log^2(n))}$. The computation time is $\mathbf{2^{O(\log^2(n))} = n^{O(\log(n))}}$.

General Space Bounds

A space bound $\mathbf{f}$ is called space-constructible iff $\mathbf{f(n) \in \Omega(\log(n))}$ and there is an algorithm using at most space $\mathbf{f(n)}$ which computes $\mathbf{f(n)}$ from an input of length $\mathbf{n}$ on the input tape.

Most common functions like $\mathbf{\log(n)}$, $\mathbf{\log^k(n)}$ for constant $\mathbf{k > 1}$, $\mathbf{\sqrt{n}}$, $\mathbf{n^k}$ for rational $\mathbf{k > 0}$ and $\mathbf{2^n}$ are space-constructible.

Theorem 4.8: Let $\mathbf{f}$ be a space-constructible function. Then any problem in NSPACE$\mathbf{(f(n))}$ is also in SPACE$\mathbf{(f^2(n))}$; the computation time is bounded from above by $\mathbf{2^{O(f^2(n))}}$.

The proof is essentially the same, one studies the graph of all possible configurations.

Polylogarithmic Space

Definition 4.9: POLYLOGSPACE be the class of all problems solvable in space $\mathbf{O(\log^k(n))}$ for some $\mathbf{k}$. As for current knowledge, both possiblitilies P = LOGSPACE and P = PSPACE are possible (of course only one of them), for the first possibility, P is a proper subclass of POLYLOGSPACE, for the second possibility, POLYLOGSPACE is a proper subclass of P. Furthermore, it could be that POLYLOGSPACE and P are incomparable.

The general belief is indeed that POLYLOGSPACE and P are incomparable and that therefore P-complete problems cannot be solved in POLYLOGSPACE. In particular, UNIFORM CONTEXT-FREE GRAMMAR MEMBERSHIP is not believed to be in POLYLOGSPACE. However, there is a slight variant of context-free grammars. That are those which are ε-free. For those the uniform membership problem is in POLYLOGSPACE.

Epsilon-Free Grammars

Definition 4.10: A context-free grammar (Σ, Δ, S, R) is ε-free if the empty word can only be derived from a nonterminal in the first step of the derivation and all other steps do not shrink the length of the current word. More precisely, if ε appears on the right side of a rule then this rule is $S \to \varepsilon$ and S does not appear on any right side of any rule.

Definition 4.11: UNIFORM EPSILON-FREE GRAMMAR MEMBERSHIP is the problem to decide for a given ε-free context-free grammar (Σ, Δ, S, R) and a word x whether the grammar generates the word x.

Theorem 4.12: UNIFORM EPSILON-FREE GRAMMAR MEMBERSHIP is in NSPACE$(\log^2(n))$ and SPACE$(\log^4(n))$.

Proof of Theorem 4.12 I

Only the NSPACE$(\log^2(n))$ algorithm will be given, the deterministic algorithm is the translation of the nondeterministic using Savitch's Theorem.

The parameter n is the maximum of number of rules, number of symbols, length of right side of any rule, length of input word x.

The nondeterministic algorithm has a basis routine $\mathrm{Verify}(u \Rightarrow^* y)$ which checks in nondeterministic $O(\log^2(n))$ space whether y can be derived from u. For this, u will be split into halves v, w and then y into halves y_v, y_w and the shorter half is done in a recursive call and the larger half will replace u and y for further processing. The splittings are nondeterministic and if there is an error, it leads to an abortion of the computation; similarly when u consists of one nonterminal then the rule which says which right side replaces u is taken nondeterministically.

Proof of Theorem 4.12 II

$\mathbf{Verify}(\mathbf{u} \Rightarrow^* \mathbf{y})$ does the following:

1. If $\mathbf{u}$ consists of a terminal word then compare whether $\mathbf{u} = \mathbf{y}$ and if yes then return TRUE else return FALSE;
2. If $\mathbf{u}$ consists of a single nonterminal $\mathbf{A}$ then pick a rule of the form $\mathbf{A} \to \mathbf{u}'$ and replace $\mathbf{u}$ by $\mathbf{u}'$ and goto 1;
3. Otherwise $\mathbf{u}$ consists of several symbols. If $|\mathbf{u}| > |\mathbf{y}|$ then return FALSE;
4. Split $\mathbf{u}$ into $\mathbf{v}, \mathbf{w}$ of same length (up to one symbol difference) and split $\mathbf{y}$ into $\mathbf{y_v}, \mathbf{y_w}$;
5. If $|\mathbf{y_v}| \leq |\mathbf{y_w}|$
   then begin call $\mathbf{Verify}(\mathbf{v} \Rightarrow^* \mathbf{y_v})$; if outcome = FALSE then return FALSE else let $\mathbf{u} = \mathbf{w}$; $\mathbf{y} = \mathbf{y_w}$; goto 1 end
   else begin call $\mathbf{Verify}(\mathbf{w} \Rightarrow^* \mathbf{y_w})$; if outcome = FALSE then return FALSE else let $\mathbf{u} = \mathbf{v}$; $\mathbf{y} = \mathbf{y_v}$; goto 1 end.

Proof of Theorem 4.12 III

Initial call is $\mathbf{Verify(S \Rightarrow^* x)}$.

Local variables in each call: $\mathbf{u, y, v, v_y, w, w_y}$. $\mathbf{u, v, w}$ stored by rule number and start and end on right side of rule or nonterminal number: $\mathbf{O(\log(n))}$. $\mathbf{y, y_v, y_w}$ stored by start and end position in $\mathbf{x}$: $\mathbf{O(\log(n))}$.

Depth of recursive calls: Each subcall marks of word half as long as $\mathbf{y}$ on right side, thus depth is at most $\mathbf{\log(n)}$ and each recursive call has at most one outgoing recursive call at a time. Overall Space usage is $\mathbf{O(\log^2(n))}$.

If $\mathbf{u = y}$ then clearly $\mathbf{u \Rightarrow^* y}$, as derivation is completed; if $\mathbf{|u| > |y|}$ one cannot do any derivation and return-value FALSE is correct; if $\mathbf{u}$ is split into $\mathbf{v, w}$ then either one can split $\mathbf{y}$ in $\mathbf{y_v, y_w}$ such that $\mathbf{(v \Rightarrow^* y_v}$ and $\mathbf{w \Rightarrow^* y_w)}$ or $\mathbf{u \not\Rightarrow^* y}$.

Problems reducible to CFL

Definition 4.13: LOGCFL is the complexity class of problems LOGSPACE reducible to the membership of one context-free language, the choice of the language might depend on the problem in LOGCFL reduced to it.

So for each language $\mathbf{L}$ in LOGCFL there is a context-free language $\mathbf{H}$ and a LOGSPACE computable function $\mathbf{f}$ such that for all $\mathbf{x}$, $\mathbf{x} \in \mathbf{L}$ iff $\mathbf{f}(\mathbf{x}) \in \mathbf{H}$.

Remark: Every context-free language is in LOGCFL and every language in LOGSPACE is in LOGCFL. Under the hypothesis P = LOGSPACE, LOGCFL = P. Under the hypothesis P = PSPACE, LOGCFL is a proper subset of P. LOGCFL is strictly contained in POLYLOGSPACE.

Example

Example 4.14: There are languages in LOGCFL which are neither context-free nor known to be in LOGSPACE. For this let $\mathbf{L}$ be a context-free language not known to be in LOGSPACE and not using the digits $\mathbf{0, 1, 2}$ and $\mathbf{H}$ be the language containing a word $\mathbf{w}$ iff $\mathbf{w}$ has the same number of $\mathbf{0, 1, 2}$ and, when deleting the digits $\mathbf{0, 1, 2}$ from $\mathbf{w}$ one obtains a word $\mathbf{v \in L}$. $\mathbf{H}$ is in LOGCFL but neither context-free nor known to be in LOGSPACE.

For seeing that $\mathbf{H}$ is in LOGCFL, one first computes with a LOGSPACE computation whether the number of all $\mathbf{0, 1, 2}$ are the same and if so, then one outputs $\mathbf{v}$ by omitting all $\mathbf{0, 1, 2}$ from $\mathbf{w}$ at the output else one outputs a fixed word not in $\mathbf{L}$.

Nick's Class

Definition 4.15: A problem $\mathbf{L}$ is in the class NC(k) iff there is a polynomial $\mathbf{p}$ and a sequence of circuits of gates with one or two input-bits such that the $\mathbf{n}$-th circuit is of size $\mathbf{p(n)}$ and determines for each $\mathbf{x} \in \{\mathbf{0}, \mathbf{1}\}^{\mathbf{n}}$ in parallel time $\mathbf{O(\log^k(n))}$ whether $\mathbf{x} \in \mathbf{L}$.

Here parallel time $\mathbf{O(\log^k(n))}$ means that there is a constant $\mathbf{k'}$ such that in each circuit all initial values are $\mathbf{0}$ and in each cycle, each value of the gate gets updated to the current outputs of the gates below and after $\mathbf{k' \cdot \log^k(n)}$ steps all gates have stabilised to the final value; in other words, the depths of the circuit is at most $\mathbf{k' \cdot \log^k(n)}$. Recall $\mathbf{\log(n) = \min\{m \geq 1 : n \leq 2^m\}}$ for this course.

There is no constraint how the circuits are obtained. LOGSPACE-uniform NC$\mathbf{(k)}$ has a LOGSPACE algorithm computing the $\mathbf{n}$-th circuit from input of length $\mathbf{n}$.

Containment I

Theorem 4.17: LOGSPACE-uniform NC$(\mathbf{k})$ is contained in SPACE$(\mathbf{O}(\log^{\mathbf{k+1}}(\mathbf{n})))$

Proof: For a proof, consider first that there is a LOGSPACE-algorithm which on input $\mathbf{m}$ finds the $\mathbf{m}$-th symbol of the description of the circuit. Furthermore, one codes each circuit starting at some position $\mathbf{m}$ as one symbol representing one of AND, OR, NOT, INP and the further positions of the bits from a lower level gate or the input separated by comma and semicolon.

The evaluation is recursive and starts with a call with position $\mathbf{0}$. Each call has subcalls if computations of lower gates are used or reads out the corresponding bit from the input. The depths of the calls is $\mathbf{O}(\log^{\mathbf{k}}(\mathbf{n}))$ and the local memory is $\mathbf{O}(\log(\mathbf{n}))$ space and thus the overall space usage is $\mathbf{O}(\log^{\mathbf{k+1}}(\mathbf{n}))$.

Containment II

Local Algorithm with input $\mathbf{m}$:
Read from position $\mathbf{m}$ onwards OPERAND and positions $\mathbf{i}$ and, in cases of OPERANDS being AND, OR, also $\mathbf{j}$.
SWITCH(OPERAND):
AND: Call algorithm with $\mathbf{i}$ and with $\mathbf{j}$ and if both results are $\mathbf{1}$ then return $\mathbf{1}$ else return $\mathbf{0}$.
OR: Call algorithm with $\mathbf{i}$ and with $\mathbf{j}$ and if both results are $\mathbf{0}$ then return $\mathbf{0}$ else return $\mathbf{1}$.
NEG: Call algortihm with $\mathbf{i}$ and if result is $\mathbf{1}$ then return $\mathbf{0}$ else return $\mathbf{1}$.
INP: Find position $\mathbf{i}$ in the input, read out the bit there and return it.

LOCAL MEMORY inside CALL is $\mathbf{m}$, $\mathbf{i}$, $\mathbf{j}$ and up to three bits (results of calls and intended return-value). This is $\mathbf{O(\log(n))}$ as used above. Note that positions in circuits of size $\mathbf{p(n)}$ need only $\mathbf{\log(p(n)) \in O(\log(n))}$ bits.

Homeworks 4.18-4.21

Show that the following problems are in NC; describe the circuits and if they are LOGSPACE uniform, explain how the algorithm outputs the circuit and describe the coding. Though NC is normally used as a class of languages (word problems), one can make the analogous definition for functions from words (coding tuples of numbers) to words (coding numbers):

Homework 4.18: Adding two positive integers in binary;

Homework 4.19: Multiplying two integers in binary (can use the addition circuits as submodules and using that they have logarithmic detph).

Homework 4.20: Computing the remainder of two positive integers in binary (first divided by second).

Homework 4.21: Computing the downrounded integer value of the fraction of two binary positive integers.

NC with Two Inputs and Two Outputs

The precise definition of Nick's Class requires not only that every input circuit uses only up to two inputs but also that only up to two outputs use it. One has then five operands: AND, OR, NEG, INP, COPY where at COPY one input is copied into two outputs which is needed to deal with situations where more than two gates use one output or input-bit. Furthermore, the description of the circuit has, for a gate at position $\mathbf{m}$, one OPERAND and four parameters which refer to the positions of the two inputs and the two outputs; for NEG and INP, the first two positions are equal, furthermore, if only one output is used, the last two parameters are equal.

Task 4.22: Using this more precise definition, show that LOGSPACE-uniform NC$(\mathbf{k})$ is contained in SPACE$(\mathbf{log^k(n)})$. Explain the algorithm in detail and how the better memorybound is achieved.

Homeworks 4.23-4.26

Homework 4.23: Show that if LOGSPACE = P then there is a constant $\mathbf{k}$ with NC = NC$(\mathbf{k})$.

Homework 4.24: Show that if P = PSPACE then all levels of the NC-hierarchy are different and P is a proper superset of NC.

Homework 4.25: Show that P = POLYLOGSPACE is impossible. For this, consider a P-complete problem $\mathbf{L}$ and use that there must be a $\mathbf{k}$ with $\mathbf{L}$ being in SPACE$(\mathbf{\log^k(n)})$. Now show that P is contained in SPACE$(\mathbf{\log^k(n)})$ as every problem in P is LOGSPACE many-one reducible to $\mathbf{L}$ and thus SPACE$(\mathbf{\log^{k+1}(n)})$ contains a problem outside P.

Homework 4.26: Prove that LOGSPACE is contained in LOGSPACE uniform NC$(\mathbf{2})$.

Lecture 5

LINSPACE and NLINSPACE are the classes of deterministic and nondeterministic linear space. The class NLINSPACE is a somewhat special complexity class, as it coincides with one level of the Chomsky hierarchy from formal languages. The Chomsky hierarchy classifies grammars generating languages (= sets of words) into regular, context-free, context-sensitive and unrestricted (= recursively enumerable). One names the classes of languages generated by the corresponding types of grammars after these classes. The regular languages coincide with CONSTANTSPACE where the only memory is the state of the Turing machine recognising the language and context-sensitive languages coincide with NLINSPACE.

So the context-sensitive languages are not only a complete problem for NLINSPACE, they coincide with this class.

Overview of Lecture 5

The main results of this lecture are the following ones:

- Formalisations of NLINSPACE: Linear bounded automata and Input-delimited space;
- Characterisation of NLINSPACE as the class of all context-senstive languages;
- LINSPACE equals determinsitic context-sensitive languages;
- Context-sensitive languages are closed under union, intersection and complementation; the same applies to all nondeterminstic space classes from logarithmic space onwards (provided that their bound is space-constructible).

Linear Bounded Automata

Definition 5.1: An linear space Turing machine is a Turing machine allowed to use space linear in the size of the input. In the case of LINSPACE, the automaton has to be deterministic; in the case of NLINSPACE, it can be nondeterministic.

More formally, the Turing machine uses on the work tape only the cells $0, 1, 2, \ldots, k \cdot n$ for some constant k depending on the machine. Furthermore, the machine can use on the work tape an arbitrary, machine-dependent alphabet.

One can, for example, compress binary symbols on the work alphabet to hexadecimal symbols and so reduce the length by a factor four. A sufficiently large work alphabet allows to have $k = 1$.

Linear Bounded Automaton

The easiest formalisation is a one-tape Turing machine with an input-word where the tape alphabet contains not only the letters of the input words but also special variants which denote that a letter is first or last of the word. Furthermore, in dependence of the machine, an arbitrary but finite additional amount of letters can be used.

For example, if the input alphabet is binary, one replaces the first bit by either $\mathbf{2}$ (if it was $\mathbf{0}$) or by $\mathbf{3}$ (if it was $\mathbf{1}$) and similarly the last ones by $\mathbf{4}$ (instead of $\mathbf{0}$) and $\mathbf{5}$ (instead of $\mathbf{1}$). If there is only one digit one uses $\mathbf{6}$ for $\mathbf{0}$ and $\mathbf{7}$ for $\mathbf{1}$. The case of the empty word is ignored. Now the Turing machine starts on $\mathbf{2/3}$ and has to move such that it never goes onto a cell which did not have an input symbol at the beginning.

Addition Machine Model

An LINSPACE addition machine with input $\mathbf{x}$ satisfies the following constraint: There is a constant $\mathbf{k}$ such that no register in the addition machine during the whole runtime of the processing of $\mathbf{x}$ takes a value larger than $(\mathbf{abs}(\mathbf{x})+\mathbf{2})^{\mathbf{k}}$ where $\mathbf{abs}(\mathbf{x})$ is the absolute value of $\mathbf{x}$. Note that the space bound of an input is logarithmic in $\mathbf{abs}(\mathbf{x})$ and thus the $\mathbf{k}$-th power uses only $\mathbf{k}$ times the space which $\mathbf{abs}(\mathbf{x})+\mathbf{2}$ uses. NLINSPACE is defined by using nondeterministic addition machines.

So three equivalent models: LBA (= Input-Deliminated One-Tape Turing Machine), LOGSPACE Turing machine, Linear Space Addition Machine. Furthermore, for both deterministic and nondetermistic computations, one has these three representations. It is conjectured but unproven that the nondeterministic variant is more powerful than the deterministic.

Context-Sensitive Languages

Definition 5.2: A grammar $(\mathbf{\Sigma}, \mathbf{\Delta}, \mathbf{S}, \mathbf{R})$ for a language $\mathbf{L}$ not containing the empty word ε with $\mathbf{R}$ being the set of rules is called context-sensitive iff every rule is of the form $\mathbf{vAw} \rightarrow \mathbf{vuw}$ with $\mathbf{A} \in \mathbf{\Delta}$, $\mathbf{v}, \mathbf{w} \in (\mathbf{\Sigma} \cup \mathbf{\Delta})^*$ and $\mathbf{u} \in (\mathbf{\Sigma} \cup \mathbf{\Delta})^+$. That is, some nonterminal $\mathbf{A}$ is replaced by a nonempty word $\mathbf{u}$ provided that in the current word, $\mathbf{v}$ is before and $\mathbf{w}$ is behind the occurrence of $\mathbf{A}$ which will be replaced. Such a replacement is therefore depending on the (perhaps empty) context $(\mathbf{v}, \mathbf{w})$ of $\mathbf{A}$ and only if this context is there, the replacement is done. Context-free grammars do not have this context-condition, but are also therefore less powerful and not closed under intersection and complement.

An equivalent type of grammar allows all rules $\mathbf{v} \rightarrow \mathbf{w}$ provided that $\mathbf{v}$ contains at least one nonterminal and $|\mathbf{v}| \leq |\mathbf{w}|$. This more liberal version will be used for proofs.

5.3: The Chomsky Hierarchy

Level 0: No restriction on grammar. This level contains all recursively enumerable languages, that is, all languages which are the range of a function computed by some algorithm without any space and time limitations.

Level 1: Context-sensitive languages. This level equals NLINSPACE (shown in this lecture).

Level 2: Context-free languages. This level contains all languages generated by context-free grammars, that is, grammars, which have rules allowing to replace some nonterminal by the given right side, no further restriction.

Level 3: Regular languages. This level equals to the languages recognised by Constant Space Turing machines and to those recognised by one-tape linear time Turing machines. Closure of finite languages under union, concatenation and Kleene star; also closed under intersection and set difference.

Example of CS Language

Example 5.4: Let $\mathbf{L}$ be the set of all nonempty words which have as many $\mathbf{0}$ as $\mathbf{1}$ as $\mathbf{2}$. Then the grammar has $\mathbf{\Delta} = \{\mathbf{S}, \mathbf{U}, \mathbf{V}, \mathbf{W}\}$ and $\mathbf{\Sigma} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ and the following rules: $\mathbf{S} \rightarrow \mathbf{UVW}$, $\mathbf{S} \rightarrow \mathbf{SUVW}$, $\mathbf{UV} \rightarrow \mathbf{VU}$, $\mathbf{VU} \rightarrow \mathbf{UV}$, $\mathbf{UW} \rightarrow \mathbf{WU}$, $\mathbf{WU} \rightarrow \mathbf{UW}$, $\mathbf{VW} \rightarrow \mathbf{WV}$, $\mathbf{WV} \rightarrow \mathbf{VW}$, $\mathbf{U} \rightarrow \mathbf{0}$, $\mathbf{V} \rightarrow \mathbf{1}$, $\mathbf{W} \rightarrow \mathbf{2}$.

The first two rules generate a word of the form $\mathbf{UVW}, \mathbf{UVWUVW}, \mathbf{UVWUVWUVW}, \ldots$ and the next six rules allow to change the order of the $\mathbf{U}, \mathbf{V}, \mathbf{W}$ in an arbitrary way and the last three rules replace every $\mathbf{U}$ by $\mathbf{0}$, every $\mathbf{V}$ by $\mathbf{1}$ and every $\mathbf{W}$ by $\mathbf{2}$. All words in the language can be generated this way and all words generated have the same number of $\mathbf{0}, \mathbf{1}, \mathbf{2}$.

Characterisation

John Myhill introduced linear bounded automata in 1960, P.S. Landweber proved some connections for the deterministic case and Sige-Yuki Kuroda proved the below theorem 1964.

Theorem 5.5 [Kuroda 1964]: A language L is context-sensitive iff it is in NLINSPACE.

The proof will be given in two parts.

(a) If the language is context-sensitive then an LBA recognises it.

(b) If an LBA with input-delimited memory (one-tape Turing machine) recognises the language L then L is context sensitive.

(a) Implication CS $\Rightarrow$ LBA

An LBA can store in the memory the current word of a derivation starting with S. In each step, it selects non-uniformly a symbol A and replaces it by the right side u, provided that the contexts v, w occur before the replacement on both sides of A and that the word does not become longer than the word to be checked. In the case that the derivation results in a word of length n which is equal to the input word x then the LBA accepts else the computation is discarded. Thus a word x is accepted by the LBA iff there is a derivation for x in the grammar — note that during the derivation, no intermediate word is longer than x, as then there is no way back to generate x. Thus whenever a replacement would make the current word strictly longer than x then one abstains from applying the corresponding rule.

(b) Implication LBA $\Rightarrow$ CS

At the beginning, the grammar generates a start configuration of the LBA with an arbitrary input word $\mathbf{x}$. A configuration looks like this:

$$(\mathbf{1}, \mathbf{b}, \mathbf{a}, -)(\mathbf{0}, \mathbf{m}, \mathbf{b}, -)(\mathbf{0}, \mathbf{m}, \mathbf{c}, \mathbf{q})(\mathbf{1}, \mathbf{e}, \mathbf{d}, -).$$

Here the first components of the configuration code the input word, here $\mathbf{1001}$. The next component informs the Turing machine about currently being at begin (b), middle (m), end (e) or only symbol of the input word (o). The third component of each symbol is the current tape symbol of the Turing machine, here $\mathbf{abcd}$, the tape alphabet can be anything depending on the machine. The last component says either that the Turing machine sits on the current (here third) symbol and has the state $\mathbf{q}$ or that it does not sit on the current symbol (default entry "$-$").

Initialisation

The initialisation creates a starting condition for the LBA. For exaple, the LBA has the start state $\mathbf{s}$ and starts on the first symbol and has not yet overwritten the input. Then the configuration would look like this:

$$(\mathbf{1},\mathbf{b},\mathbf{1},\mathbf{s})(\mathbf{0},\mathbf{m},\mathbf{0},-)(\mathbf{0},\mathbf{m},\mathbf{0},-)(\mathbf{1},\mathbf{e},\mathbf{1},-).$$

So the part of the grammar which produces this situation has these rules: $\mathbf{S}\to\mathbf{T}(\mathbf{1},\mathbf{e},\mathbf{1},-)$, $\mathbf{S}\to\mathbf{T}(\mathbf{0},\mathbf{e},\mathbf{0},-)$, $\mathbf{T}\to\mathbf{T}(\mathbf{0},\mathbf{m},\mathbf{0},-)$, $\mathbf{T}\to\mathbf{T}(\mathbf{1},\mathbf{m},\mathbf{1},-)$, $\mathbf{T}\to(\mathbf{1},\mathbf{b},\mathbf{1},\mathbf{s})$, $\mathbf{T}\to(\mathbf{0},\mathbf{b},\mathbf{0},\mathbf{s})$. So the input word is any binary word and the Turing machine starts with the same word on its Turing tape and the state $\mathbf{s}$ on the first symbol of the input. In the next slide come the rules for working on the tape. The rules for the initialisation do not use any context. The nonterminals $\mathbf{S},\mathbf{T}$ are only used in this phase.

Main Phase

Assume now that the Turing machine sits on the symbol $(\mathbf{0}, \mathbf{m}, \mathbf{a}, \mathbf{q})$ with state $\mathbf{q}$ and neighbour $(\mathbf{1}, \mathbf{m}, \mathbf{b}, -)$ and goes to the to a state $\mathbf{r}$ on the next symbol after writing a $\mathbf{c}$ onto the Turing tape before leaving. To allow this, the following rules must be in the rule set of the grammar:

$$(\mathbf{0}, \mathbf{m}, \mathbf{a}, \mathbf{q})(\mathbf{1}, \mathbf{m}, \mathbf{b}, -) \rightarrow (\mathbf{0}, \mathbf{m}, \mathbf{c}, -)(\mathbf{1}, \mathbf{m}, \mathbf{b}, \mathbf{r}).$$

Each allowed transition of the LBA can be coded into such a rule. Furthermore, one assumes that after the computation, if the LBA wants to accept, it goes to the first symbol and then takes an accepting state $\mathbf{a}$. The configuration looks as follows:

$$(\mathbf{1}, \mathbf{b}, \mathbf{e}, \mathbf{a})(\mathbf{0}, \mathbf{m}, \mathbf{f}, -)(\mathbf{0}, \mathbf{m}, \mathbf{g}, -)(\mathbf{1}, \mathbf{e}, \mathbf{h}, -).$$

Only the third and fourth component of symbols change during the simulation.

Termination

If an accepting state is reached, the Turing machine applies the following rules.

$$(1, b, i, a) \to 1, (0, b, j, a) \to 0.$$

These rules can have anything at entries i, j. Once this is done, there are rules to convert from the front to the end each nonterminal into a terminal, here $\Sigma = \{0, 1\}$.

$$\begin{aligned}
&0(0, m, i, -) \to 00, 0(1, m, i, -) \to 01, \\
&1(0, m, i, -) \to 10, 1(1, m, i, -) \to 11, \\
&0(0, e, i, -) \to 00, 0(1, e, i, -) \to 01, \\
&1(0, e, i, -) \to 10, 1(1, e, i, -) \to 11.
\end{aligned}$$

These rules allow to convert the whole word into a terminal word equal to the originally guessed input x of the LBA provided that the simulation of the LBA leads to acceptance.

Deterministic CS Grammars

One might ask how to put determinism into this concept. As the generation of words starting from S has multiple outcomes, grammars are nondeterministic by definition. However, one might ask how much information is needed to know which rules to apply to derive a specific word x. The following is a possible definition for a deternistic context-sensitive grammar and language.

Definition 5.6: A context-sensitive grammar is called deterministic iff there is a LOGSPACE computable function which computes from inputs x and y where x is the word to be generated and y is the current word of the derivation, which rule to apply and at what position of y. A language is deterministic context-sensitive iff it has a deterministic context-sensitive grammar.

Characterising DCS Languages

Theorem 5.7: A language is deterministic context-sensitive if and only if a deterministic LBA can recognise it.

The LOGSPACE guidance function tells which rule to apply in a derivation. Initially $y = S$ and x is the given input word. In every step, the LOGSPACE function says which rule to apply to y in order to update the current word to the next step. If it ever happens that $x = y$ then the LBA accepts. If the derivation takes longer than $(|\Sigma| + |\Delta| + 1)^n$ steps where $n = |x|$ or ends in a rejecting state then the deterministic LBA rejects.

LINSPACE $\Rightarrow$ DCS

Now assume that a deterministic LBA is given to witness that the language is in LINSPACE. The grammar constructed from the LBA is the same as in Theorem 5.1 (b). Now the LOGSPACE guiding function does the following:

First it applies those rules using the nonterminals $\mathbf{S}, \mathbf{T}$ to generate

$$(\mathbf{x_1}, \mathbf{b}, \mathbf{x_1}, \mathbf{s})(\mathbf{x_2}, \mathbf{m}, \mathbf{x_2}, -) \ldots (\mathbf{x_{n-1}}, \mathbf{m}, \mathbf{x_{n-1}}, -)(\mathbf{x_n}, \mathbf{e}, \mathbf{x_n}, -)$$

from $\mathbf{x} = \mathbf{x_1 x_2 \ldots x_n}$. Then the LOGSPACE guiding function determines at every situation which rule of the Turing machine applies (it is unique) and the position of the rule is that of the head and one neighbouring field (depending on the direction of the upcoming move). If then the LBA accepts then the last phase applies and one can see from the rules that always exactly one rule applies.

5.8: The LBA Problems

It is easy to see that if one takes the union or intersection of two languages recognised by LBAs, one again has a language recognised by an LBA and if the given two LBAs are deterministic, so are those for union and intersection. Kuroda asked [1964] the following two questions.

(LBA 1) Is there, for every nondeterministic LBA, an equivalent deterministic LBA? In other words, do context-sensitive and deterministic context-sensitive languages coincice? Is NLINSPACE = LINSPACE?

(LBA 2) If some LBA recognises a language, can its complement also be recognised by an LBA? In other words, are context-sensitive languages closed under complement?

Question (LBA 1) is still open, Question (LBA 2) was open for over twenty years until Immerman and Szelepcsényi solved it independetly at the same time.

Context-Sensitive Languages

Theorem 5.9 [Immerman and Szelepcsényi 1987]
The complement of a context-sensitive language is context-sensitive.

Representation of $\Sigma^* - L$
The complent of L will be enumerated by an LBA which has constantly many word-variables of the same length as the input x and which accepts x if it does not find a derivation for x in the original grammar and has verified by nondeterministic counting that all derivations have been taken care off.

Proof-Method of Nondeterministic Counting
If i strings can be derived in ℓ steps then one can nondeterministically check which string y can be derived in $\ell + 1$ steps and count their number j.

Basic Algorithm

1. For any $\mathbf{x} \in \mathbf{\Sigma}^+$, try to verify $\mathbf{x} \notin \mathbf{L}$ as follows;
2. Let $\mathbf{u}$ be the length-lexicographically largest string in $(\mathbf{N} \cup \mathbf{\Sigma})^{|\mathbf{x}|}$;
3. Let $\mathbf{i} = \mathbf{Succ_{ll}}(\varepsilon)$;
4. For $\ell = \varepsilon$ to $\mathbf{u}$ Do Begin
5. Let $\mathbf{j} = \varepsilon$;
6. For all $\mathbf{y} \leq_{\mathbf{ll}} \mathbf{u}$ Do Begin
7. Derive words $\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_i}$ non-deterministically in length-lexicographic order in up to ℓ steps each and check:
8. If some $\mathbf{w_m}$ satisfies $\mathbf{w_m} \Rightarrow \mathbf{y}$ or $\mathbf{w_m} = \mathbf{y}$ then let $\mathbf{j} = \mathbf{Succ_{ll}}(\mathbf{j})$;
9. If some $\mathbf{w_m}$ satisfies $\mathbf{w_m} = \mathbf{x}$ or $\mathbf{w_m} \Rightarrow \mathbf{x}$ then abort computation (as $\mathbf{x} \in \mathbf{L}$); End (of For-Loop 6);
10. Let $\mathbf{i} = \mathbf{j}$; let $\mathbf{j} = \varepsilon$; End (of For-Loop 4);
11. If the algorithm has not been aborted then $\mathbf{x} \notin \mathbf{L}$.

Refined Algorithm I

1: Choose an $\mathbf{x} \in \mathbf{\Sigma}^+$ and initial all other variables as ε;

2: Let $\mathbf{u} = (\mathbf{max}_{\mathbf{ll}}(\mathbf{N} \cup \mathbf{\Sigma}))^{|\mathbf{x}|}$;

3: Let $\mathbf{i} = \mathbf{Succ}_{\mathbf{ll}}(\varepsilon)$ and $\ell = \varepsilon$;

4: While $\ell <_{\mathbf{ll}} \mathbf{u}$ Do Begin

  5: Let $\mathbf{j} = \varepsilon$;

  6: Let $\mathbf{y} = \varepsilon$;

  7: While $\mathbf{y} <_{\mathbf{ll}} \mathbf{u}$ Do Begin

    8: Let $\mathbf{y} = \mathbf{Succ}_{\mathbf{ll}}(\mathbf{y})$;

    9: $\mathbf{h} = \varepsilon$ and $\mathbf{w} = \varepsilon$;

    10: While $\mathbf{h} <_{\mathbf{ll}} \mathbf{i}$ Do Begin

      11: Nondeterministically replace $\mathbf{w}$ by $\mathbf{w}'$ with $\mathbf{w} <_{\mathbf{ll}} \mathbf{w}' \leq_{\mathbf{ll}} \mathbf{u}$;

      12: Let $\mathbf{v} = \mathbf{S}$;

      13: Let $\mathbf{k} = \varepsilon$;

Refined Algorithm II

14: While $(\mathbf{v} \neq \mathbf{w}) \wedge (\mathbf{k} <_{\mathbf{ll}} \ell)$ Do Begin

  15: Nondeterministically replace $(\mathbf{k}, \mathbf{v})$ by $(\mathbf{k}', \mathbf{v}')$ with $\mathbf{k} <_{\mathbf{ll}} \mathbf{k}'$ and $\mathbf{v} \Rightarrow \mathbf{v}'$ End (of While in 14);

16: If $\mathbf{v} \neq \mathbf{w}$ Then abort the computation;

17: If $\mathbf{w} = \mathbf{x}$ or $\mathbf{w} \Rightarrow \mathbf{x}$ Then abort the computation;

18: If $\mathbf{w} \neq \mathbf{y}$ and $\mathbf{w} \not\Rightarrow \mathbf{y}$

  19: Then Let $\mathbf{h} = \mathbf{Succ}_{\mathbf{ll}}(\mathbf{h})$

  20: Else Let $\mathbf{h} = \mathbf{i}$

21: End (of While in 10);

22: If $\mathbf{w} = \mathbf{y}$ or $\mathbf{w} \Rightarrow \mathbf{y}$ Then $\mathbf{j} = \mathbf{Succ}_{\mathbf{ll}}(\mathbf{j})$

23: End (of While in 7);

24: Let $\mathbf{i} = \mathbf{j}$;

25: Let $\ell = \mathbf{Succ}_{\mathbf{ll}}(\ell)$ End (of While in 4);

26: If the algorithm has not yet aborted Then generate $\mathbf{x}$.

Verification of LBA

For the LBA, numbers are strings in length-lexicographic order.

The LBA accepts $\mathbf{x}$ only if the nondeterministic counting has counted up all words in $(\mathbf{\Sigma} \cup \mathbf{\Delta})^*$ of length up to $\mathbf{n} = |\mathbf{x}|$ and as the word ε does not occur in the derivation, every longer derivation has a repetition and is thus not the shortest derivation. If there is a derivation accepting $\mathbf{x}$ of length ℓ then the algorithm would have detected it for any ℓ up to length $\mathbf{n}$. So $\mathbf{x}$ is accepted iff it is verified that such a derivation does not exist.

The algorithm stores constantly many variables all taking as value a word up to length $\mathbf{n}$ over the alphabet $\mathbf{\Sigma} \cup \mathbf{\Delta}$. One can use a sufficiently large alphabet – in this alphabet each symbol is a tuple with one component being the input word at this position and all other components being the symbol of the corresponding variable at the corresponding position.

Homework 5.10-5.14

Provide Context-sensitive Grammars for the following languages and show that they are also in LOGSPACE (most context-sensitive languages are not).

Homework 5.10: Do this with the language of all words $0^n 1^n 2^n$ where $n \geq 2$.

Homework 5.11: Do this with the language of all words of the form uu where u is some nonempty word over $\{0, 1, 2\}$.

Homework 5.12: Do this with the language of all words of the form uu where u is some nonempty palindrome over $\{0, 1, 2\}$.

Homework 5.13: Do this with the language of all words $0^n 1^{2n} 2^{3n}$ where $n \geq 2$.

Homework 5.14: Do this with the language of all words in $\{0, 1, 2\}^n$ where there are distinct $i, j \in \{0, 1, 2\}$ which do not occur in the word the same amount of times.

Homework 5.15-5.18

Let $\phi(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}, \mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_n})$ be a formula consisting of a conjunction of clauses where every clause is a disjunction of literals and every literal is either constant $\mathbf{0}$ or constant $\mathbf{1}$ or some variable $\mathbf{x_k}$ or $\mathbf{y_k}$ or some negated variablable $\neg\mathbf{x_k}$ or $\neg\mathbf{y_k}$. There are assumed brackets to be around each clause (as conjunction binds normally more than disjunction). Write deterministic LINSPACE algorithms to evaluate the following formulas. Say whether the formula is definitely in P or NP or CoNP (without any assumption).

Homework 5.15: $\exists\mathbf{x_1}\ldots\exists\mathbf{x_n}\exists\mathbf{y_1}\ldots\exists\mathbf{y_n}\,[\phi(\mathbf{x_1},\ldots,\mathbf{y_n})]$.

Homework 5.16: $\exists\mathbf{x_1}\ldots\exists\mathbf{x_n}\forall\mathbf{y_1}\ldots\forall\mathbf{y_n}\,[\phi(\mathbf{x_1},\ldots,\mathbf{y_n})]$.

Homework 5.17: $\exists\mathbf{x_1}\forall\mathbf{y_1}\exists\mathbf{x_2}\forall\mathbf{y_2}\ldots\exists\mathbf{x_n}\forall\mathbf{y_n}\,[\phi(\mathbf{x_1},\ldots,\mathbf{y_n})]$.

Homework 5.18: $\forall\mathbf{x_1}\ldots\forall\mathbf{x_n}\exists\mathbf{y_1}\ldots\exists\mathbf{y_m}\,[\phi(\mathbf{x_1},\ldots,\mathbf{y_m})]$ where $\mathbf{m} = \mathbf{log(n)} = \mathbf{min}\{\mathbf{m} \geq \mathbf{1} : \mathbf{2^m} \geq \mathbf{n}\}$.

Lecture 6

Lecture 6 will address the nondeterministic time complexity class NP and its relation to related classes P and its complement CoNP.

The motivation for the main class NP is that in mathematics (as one might remember from school) there are many tasks where it is quite difficult to find a solution while it is easy to verify that the found solution is correct by testing.

For nondeterministic machines, it might be difficult to find out whether there is a nondeterministic computation leading to ACCEPT, but if one knows for each branching which decision the machine takes to reach ACCEPT then one can easily verify that the machine accepts the input.

More concrete problems in NP, in particular complete ones, are provided in this lecture.

P, NP and CoNP

Definition 6.1: A Nondeterministic Polynomial Time Machine $\mathbf{M}$ is a Turing machine (or addition machine) with nondeterministic branchings which on input words $\mathbf{x}$ runs on all computations at most time $\mathbf{f(n)}$ for inputs of size $\mathbf{n}$ where $\mathbf{f}$ is a polynomial and which might reach on some inputs ACCEPT or REJECT. Furthermore $\mathbf{M}$ is good iff there is no input on which both ACCEPT and REJECT can be reached.

A problem $\mathbf{L}$ is in NP iff some nondeterministic polynomial time machine $\mathbf{M}$ can on every $\mathbf{x} \in \mathbf{L}$ but on no $\mathbf{x} \notin \mathbf{L}$ reach ACCEPT.

A problem $\mathbf{L}$ is in CoNP iff some nondeterministic polynomial time machine $\mathbf{M}$ can on every $\mathbf{x} \notin \mathbf{L}$ but on no $\mathbf{x} \in \mathbf{L}$ reach REJECT.

A problem $\mathbf{L}$ is in NP $\cap$ CoNP iff there is a good nondeterministic polynomial time machine $\mathbf{M}$ which can reach on all $\mathbf{x} \in \mathbf{L}$ ACCEPT and on all $\mathbf{x} \notin \mathbf{L}$ REJECT.

Complete Problems

Definition 6.2: A problem $\mathbf{L}$ is complete for a complexity class $\mathbf{C}$ iff for every further problem $\mathbf{H}$ in $\mathbf{C}$ there is a LOGSPACE computable function $\mathbf{f}$ such that for all $\mathbf{x}$, $\mathbf{H}(\mathbf{x}) = \mathbf{L}(\mathbf{f}(\mathbf{x}))$, that is, $\mathbf{x} \in \mathbf{H}$ if and only if $\mathbf{f}(\mathbf{x}) \in \mathbf{L}$.

Example 6.3: A complete problem for CoNP is whether a nondeterministic polynomial time machine is good on some length $\mathbf{n}$. So the input is a nondeterministic polynomial time machine $\mathbf{M}$ and a time bound (polynomial) $\mathbf{f}$ and a length $\mathbf{n}$ (given in unary) and the machine is good at length $\mathbf{n}$ iff there is no input $\mathbf{x}$ of length $\mathbf{n}$ on which the machine either has a run taking longer than $\mathbf{f}(\mathbf{n})$ steps or has two runs one of them reaching ACCEPT and one reaching REJECT.

Note that the length $\mathbf{n}$ must be part of the input; otherwise the problem is outside CoNP and even undecidable, that is, there is no algorithm at all which solves this problem.

Complexity Parameters

Main parameters of certain problem types.

Problem Type	Parameter	Name
Graphs	Number of nodes	$\mathbf{n}$
	Number of edges	$\mathbf{m}$
Languages	Alphabet Size	$\mathbf{\|\Sigma\|}$
	Length of Word	$\mathbf{n}$
Formulas and Circuits	Number of variables	$\mathbf{n}$
	Number of clauses / gates	$\mathbf{m}$
	Overall number of literals	ℓ
	Maximal clause length	$\mathbf{k}$
	Depth or height	$\mathbf{h}$
Natural Numbers	Size $\mathbf{\min\{k \geq 1 : x \leq 2^k\}}$	$\mathbf{n}$
Tuples and Matrices	Size	$\mathbf{n, m \times n}$

Satisfiability

Literal = Variable, Negated Variable, Truth Constant;
Clause = Disjunction (Or) over literals;
Formula = Conjunction (And) over clauses.
Monotone = No use of Negation.

Example: Let ϕ be
$(\mathbf{x_1} \vee \mathbf{x_2}) \wedge (\mathbf{x_1} \vee \neg \mathbf{x_3}) \wedge (\mathbf{x_2} \vee \mathbf{x_4}) \wedge (\mathbf{x_1} \vee \mathbf{x_4}) \wedge (\mathbf{x_1} \vee \mathbf{x_3}) \wedge (\mathbf{0} \vee \mathbf{1})$.
Now $\mathbf{n} = \mathbf{4}$, $\mathbf{m} = \mathbf{6}$, $\ell = \mathbf{12}$, $\mathbf{k} = \mathbf{2}$ and ϕ is not monotone.

The formula is in CNF (Conjunctive Normal Form, always assumed for formulas) and is satisfiable, as one can set $\mathbf{x_1} = \mathbf{1}$ and $\mathbf{x_4} = \mathbf{1}$ (makes all clauses true).

SAT = Set of all CNF formulas which are satisfiable (standard NP-complete problem). $\mathbf{3}$SAT = Set of all CNF formulas which are satisfiable and where $\mathbf{k} \leq \mathbf{3}$, similarly $\mathbf{2}$SAT and $\mathbf{4}$SAT. XSAT = Set of all CNF formulas which are satisfiable in a way that for each clause exactly one literal is true.

NP-completeness of 3SAT I

Theorem 6.4: 3SAT is NP-complete.

Given a SAT CNF formula with parameters $\mathbf{n}$ and $\mathbf{m}$ and ℓ, one introduces for each clause with $\mathbf{k}$ literals, where $\mathbf{k} \geq 4$, $\mathbf{k-3}$ additional new variables to split it into $\mathbf{3SAT}$ clauses as in the following example:

$\mathbf{x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5}$ is split into
$\mathbf{x_1 \vee x_2 \vee y_1}$, $\mathbf{\neg y_1 \vee x_3 \vee y_2}$ and $\mathbf{\neg y_2 \vee x_4 \vee x_5}$.

If now first clause is satisfied, say by $\mathbf{x_3}$ is true, then one can also satisfy the three second clauses by letting $\mathbf{y_1}$ and $\mathbf{\neg y_2}$ be true. Note that here $\mathbf{5-3=2}$ new variables were inserted. However if the first clause is not satisfied, then choosing $\mathbf{y_1, y_2}$ does not allow to make all three new clauses true, as among the literals of $\mathbf{y_1, y_2}$, only two can be true, thus one of $\mathbf{x_1 \vee x_2,\ x_3,\ x_4 \vee x_5}$ must be made true to have all three true.

NP-Completeness of 3SAT II

The previous slide explained how to replace an $\mathbf{n}$-var $\mathbf{m}$-clause ℓ-literal SAT formula by a 3SAT formula with up to $\mathbf{n} + \ell - \mathbf{3m}$ variables and up to $\ell - \mathbf{2m}$ clauses. If the previous formula was in $\mathbf{k}$SAT then $\ell \leq \mathbf{km}$ so that the new formula has up to $\mathbf{n} + (\mathbf{k} - \mathbf{3})\mathbf{m}$ variables and up to $(\mathbf{k} - \mathbf{2})\mathbf{m}$ clauses.

The algorithm can be carried out in LOGSPACE, one just needs a secure method to generate new variable names and then can translate clause by clause with a three-tape LOGSPACE Turing machine. The old variable names are kept and the new ones are just used twice for each splitting.

2SAT is in P

Theorem 6.5: 2SAT is in P.

For the polynomial time algorithm on the next slide, note that in the case of two literals per clause, α, β have both at most one literal and only new two literal clauses are created. Furthermore, the number of clauses is bounded by $\mathbf{4n^2 + 2n}$ throughout the algorithm, so all data fits in polynomial space. Each processing of a variable goes in polynomial time, as one can check each condition with a scan over the ocurrences of the variables in the input.

Furthermore, the same polynomial time algorithm applies if every variable occurs at most two times, independently of the size of the clauses. Thus 2OCCURSAT is also in P.

3OCCURSAT is NP-complete, as one can replace one variable $\mathbf{x}$ occurring $\mathbf{k}$ times by $\mathbf{y_1, y_2, \ldots, y_k}$, respectively, and then add clauses for $\mathbf{y_1 \rightarrow y_2, y_2 \rightarrow y_3, \ldots, y_k \rightarrow y_1}$.

Algorithm for 2SAT is in P

Algorithm 6.6: For each variable $\mathbf{x_k}$ still occuring in some clause, do the first case which applies; if no clause is left, return SATISFIABLE.

1. If there are two single literal clauses $\mathbf{x_k}$ and $\neg\mathbf{x_k}$ then return UNSATISFIABLE.

2. If either $\mathbf{x_k}$ occurs only positively or there is a single literal clause containing $\mathbf{x_k}$ then remove all clauses containing $\mathbf{x_k}$ and remove $\neg\mathbf{x_k}$ from all clauses where it occurs (due to 1. done before, such clauses still have other literals).

3. Do the action corresponding to 2. if either $\mathbf{x_k}$ occurs only negatively or there is a single literal clause $\neg\mathbf{x_k}$.

4. For all pairs of clauses $\alpha \vee \mathbf{x_k}$, $\beta \vee \neg\mathbf{x_k}$ put the clause $\alpha \vee \beta$ into the set of clauses and then remove all clauses containing $\mathbf{x_k}$ or $\neg\mathbf{x_k}$. This is called resolution.

Weighted 2SAT

A weight-function assigns to every variable $\mathbf{x}$ weights $\mathbf{f(x)}$ and $\mathbf{f(\neg x)}$ which are nonnegative integers. A WEIGHTED2SAT instance is given by a set of $\mathbf{2}$SAT clauses and a weight function and a target value $\mathbf{k}$. Now a WEIGHTED2SAT instance with variables $\mathbf{x_1, \ldots, x_n}$ is in WEIGHTED2SAT iff there is an assignment of binary values $\mathbf{z_k}$ to variables $\mathbf{x_k}$ such that the sum over all $\mathbf{f(x_k)}$ with $\mathbf{z_k = 1}$ and $\mathbf{f(\neg x_k)}$ with $\mathbf{z_k = 0}$ is $\mathbf{k}$; for the ease of notation one writes $\mathbf{f(z_1), \ldots, f(z_n)}$ for these values.

Furthermore WEIGHTED2SAT$_\mathbf{c}$ is the same problem with the additional constraint that all weights are at most $\mathbf{c}$.

Theorem 6.7: WEIGHTED2SAT and, for all $\mathbf{c \geq 1}$, also WEIGHTED2SAT$_\mathbf{c}$ are NP-complete.

Membership in NP is trivial; just guess the $\mathbf{z_k}$ if they exist.

NP-hardness Construction

One translates XSAT into WEIGHTED2SAT. For this, one adds for each clause with literals $\mathbf{x_1, x_2, \ldots, x_h}$ (they can be negated variables) all conditions $\neg\mathbf{x_i} \vee \neg\mathbf{x_j}$ with $\mathbf{1 \leq i < j \leq h}$ into the 2SAT instance to be constructed.

Furthermore, consider a variable $\mathbf{x}$ which occurs in $\mathbf{i}$ clauses as $\mathbf{x}$ and in $\mathbf{j}$ clauses as $\neg\mathbf{x}$. Then one sets $\mathbf{f(x) = i}$ and $\mathbf{f(\neg x) = j}$. The number $\mathbf{k}$ is the overall number of XSAT clauses in the XSAT instance.

In the case that one has WEIGHTED2SAT$_\mathbf{1}$, one adds to each variable $\mathbf{x}$, which occurs $\mathbf{i}$ times positively and $\mathbf{j}$ times negatively, $\mathbf{h = \max\{i, j\}}$ further variables $\mathbf{y_1, \ldots, y_h}$ and adds to the 2SAT clauses for $\mathbf{x \rightarrow y_1}$, $\mathbf{y_1 \rightarrow y_2}$, ..., $\mathbf{y_h \rightarrow x}$ and one defines $\mathbf{f(x) = 0}$, $\mathbf{f(\neg x) = 0}$, $\mathbf{f(y_\ell)}$ to be $\mathbf{1}$ if $\ell \leq \mathbf{i}$ and to be $\mathbf{0}$ if $\ell > \mathbf{i}$, $\mathbf{f(\neg y_\ell)}$ to be $\mathbf{1}$ if $\ell \leq \mathbf{j}$ and to be $\mathbf{0}$ if $\ell > \mathbf{j}$.

Ideas of Verification

Assume that $z_1, \ldots, z_n$ is a satisfying assignment of underlying XSAT instance and that one extends f to assignments as indicated above. Then $z_1, \ldots, z_n$ satisfies the derived 2SAT instance. Furthermore, $f(z_h)$ is the number of clauses made true by z_h and, as the XSAT assignment is satisfying, the sum of the $f(z_h)$ is k.

For the converse way, if $z_1, \ldots, z_n$ satisfies the derived 2SAT formula and $\sum_h f(z_h) = k$ then the 2SAT formula enforces that each clause has at most one satisfied literal and the formula $\sum_h f(z_h) = k$ enforces that the overall number of satisfied literals is k; thus each clause has exactly one and the given assignment is an XSAT assignment.

The third item just tells how to code with additional variables which are all equal to the main variable in the case that the weight bound 1 is there.

Trivial SAT Algorithm

One can just test out all 2^n combinations of the variables and for each check whether the assignment is satisfying. Thus the runtime is at most $2^n \cdot Poly(n+m)$. If there are additional constraints, it goes slightly better.

Theorem 6.8. kSAT can be solved in time $((2^k - 1)^{1/k})^n \cdot Poly(n+m)$.

Algorithm: Pick the shortest clause (with length k') and branch all the variable-combinations of the variables in this clause satisfying it. Each branch removes k' variables; for all clauses where these variables occur, if the values of these variables make the clause true then remove the clause (as satisfied) else remove the literals of these variables from the clause. If empty unsatisfied clauses remain, the corresponding possibility is terminated (as not satisfying). Repeat this until all variables are used up.

Evaluation I

$\mathbf{k}$SAT: If $\mathbf{k}' \leq \mathbf{k}$ then $(\mathbf{2}^{\mathbf{k}'} - \mathbf{1})^{\mathbf{1}/\mathbf{k}'} \leq (\mathbf{2}^{\mathbf{k}} - \mathbf{1})^{\mathbf{1}/\mathbf{k}}$. Thus algorithm in Time $((\mathbf{2}^{\mathbf{k}} - \mathbf{1})^{\mathbf{1}/\mathbf{k}})^{\mathbf{n}} \cdot \mathbf{Poly}(\mathbf{n}+\mathbf{m})$. This is time $\mathbf{1.91294}^{\mathbf{n}} \cdot \mathbf{Poly}(\mathbf{n}+\mathbf{m})$ for $\mathbf{3}$SAT.

For this one has to verify that all $\mathbf{k} \geq \mathbf{1}$ satisfy $(\mathbf{2}^{\mathbf{k}} - \mathbf{1})^{\mathbf{1}/\mathbf{k}} < (\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{1})^{\mathbf{1}/(\mathbf{k}+\mathbf{1})}$. Note that one can raise both sides to the power $\mathbf{k}(\mathbf{k}+\mathbf{1})$ without changing the relation and one has $(\mathbf{2}^{\mathbf{k}} - \mathbf{1})^{\mathbf{k}+\mathbf{1}} < (\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{1})^{\mathbf{k}}$. To see this, one takes the logarithm which is monotone on positive numbers (here no rounding is done). So the above holds iff the following holds: $(\mathbf{k}+\mathbf{1}) \cdot \mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) < \mathbf{k} \cdot \mathbf{log}(\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{1})$.

Now one has $(\mathbf{k}+\mathbf{1})\mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) < \mathbf{k} \cdot \mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) + \mathbf{k} = \mathbf{k} \cdot (\mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) + \mathbf{1}) < \mathbf{k} \cdot \mathbf{log}(\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{1})$ as $\mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) < \mathbf{k}$ and $\mathbf{log}(\mathbf{2}^{\mathbf{k}} - \mathbf{1}) + \mathbf{1} = \mathbf{log}(\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{2}) < \mathbf{log}(\mathbf{2}^{\mathbf{k}+\mathbf{1}} - \mathbf{1})$.
Throughout the lecture, the logarithm has base $\mathbf{2}$.

Evaluation II

X$\mathbf{k}$SAT: One needs only to consider $\mathbf{k}$ in place of $\mathbf{2^k - 1}$ possibilities, as exactly one out of the $\mathbf{k}$ literals is $\mathbf{1}$ and all others are $\mathbf{0}$ for satisfying the clause. Clause-length $\mathbf{3}$ is worst case for this algorithm and time is $\mathbf{1.44225^n \cdot Poly(n+m)}$ for all X$\mathbf{k}$SAT and XSAT in general.

For both problems, much better algorithms are known. Paturi, Pudlák, Saks and Zane [2005] showed that $\mathbf{3}$SAT can be done by a randomised algorithm in time $\mathbf{1.30685^n \cdot Poly(n+m)}$ and Liu [2018] provided a deterministic algorithm in time $\mathbf{O(n^{1.3280})}$. Wahlström [2007] provided an $\mathbf{O(1.0984^n)}$ algorithm for X3SAT and Gordon Hoi (2020) showed that XSAT can be done in time $\mathbf{1.1674^n \cdot Poly(n+m)}$ by a deterministic algorithm.

Graph Problems

Definition 6.9: The exact clique problem is given as the set of all (V, E, k) where V is a set of nodes and E is a set of edges (unordered pairs of nodes) between the nodes of V and k is an integer such that an instance (V, E, k) is in EXACTCLIQUE iff the graph (V, E) has a clique with exactly k nodes. Here a clique is a subset of V such that each pair of nodes in the subset is connected by an edge.

Theorem 6.10: Exact Clique is NP-complete.

Membership in EXACTCLIQUE can be verified by guessing a clique of size k (listing out the nodes explicitly) and a LOGSPACE algorithm then can check that the listed out subset has k members and each two members of it are connected.

Negation-Free XSAT

Recall that an XSAT instance is given as a set of variables $\mathbf{X}$ and set of clauses $\mathbf{Y}$ using these variables $\mathbf{X}$. Now the instance $(\mathbf{X}, \mathbf{Y})$ is in XSAT iff there truth-assignment to the variables in $\mathbf{X}$ which makes in each clause exactly one literal true. XSAT is NP-complete.

NEGATIONFREEXSAT instances are XSAT instances in which all literals are only variables and no negated variable occurs in a clause; NEGATIONFREEXSAT is NP-complete.

To see this, one translates an XSAT instance into a negation-free XSAT instance as follows: If a variable occurs only positively, it just remains as it is. If a variable $\mathbf{x}$ occurs only negatively, one replaces each literal $\neg\mathbf{x}$ by $\mathbf{x}$ without changing solvability. If a variable $\mathbf{x}$ occurs both positively and negatively, one introduces a new variable $\mathbf{y}$ added replaces in the clauses all $\neg\mathbf{x}$ by $\mathbf{y}$ and then one adds the new clause $\mathbf{x} \vee \mathbf{y}$ to the instance.

Coding XSAT into Cliques

An instance of NEGATIONFREEXSAT is now translated into an instance of EXACTCLIQUE as follows.

Let $\mathbf{V}$ be the set of all pairs $(\mathbf{i},\mathbf{j})$ of numbers such that the $\mathbf{i}$-th variable occurs in the $\mathbf{j}$-th clause. Let $\mathbf{E}$ contain all unordered pairs ($\mathbf{2}$-element-sets) $\{(\mathbf{i},\mathbf{j}),(\mathbf{i}',\mathbf{j}')\}$ where either $\mathbf{i}=\mathbf{i}'$ and $\mathbf{j}\neq\mathbf{j}'$ or $\mathbf{i}\neq\mathbf{i}'$ and there is no $\mathbf{j}''$ such that $(\mathbf{i},\mathbf{j}''),(\mathbf{i}',\mathbf{j}'')$ are both in $\mathbf{V}$. Now $\mathbf{k}$ is the number of clauses.

Now for every solution of the NEGATIONFREEXSAT instance with a solution to the variables, the clique of size $\mathbf{k}$ consists of all nodes $(\mathbf{i},\mathbf{j})$ in $\mathbf{V}$ where the $\mathbf{i}$-th variable is set $\mathbf{1}$. The size of this clique is $\mathbf{k}$ as for each clause exactly one node $(\mathbf{i},\mathbf{j})$ goes into this clique. As no clique contains nodes $(\mathbf{i},\mathbf{j}),(\mathbf{i}',\mathbf{j}')$ such that $\mathbf{i},\mathbf{i}'$ are different variables sharing a clause, each clique codes a partial solution; the coded solution is a full solution iff its size is $\mathbf{k}$.

Knapsack I

Definition 6.11: A list $\mathbf{x_1, x_2, \ldots, x_n, y}$ of natural numbers is in KNAPSACK iff there is a binary list $\mathbf{z_1, z_2, \ldots, z_n}$ with $\sum_{\mathbf{k=0,\ldots,n}} \mathbf{x_k z_k = y}$. Size parameter $\mathbf{n + \log\max\{x_1, \ldots, x_n, y\}}$.

Theorem 6.12: KNAPSACK is NP-complete.

For membership in NP, one just guesses $\mathbf{z_1, \ldots, z_n}$. The homeworks will show that not only XSAT but also X3SAT is NP-complete. Similarly NEGATIONFREEX3SAT is NP-complete. Given an instance, let $\mathbf{n}$ be the number of its variables and $\mathbf{m}$ be the number of its clauses. Now one assigns to each variable $\mathbf{\hat{x}_i}$ of a negation-free X3SAT instance the number $\mathbf{x_i} = \sum_{\mathbf{j:\hat{x}_i\ occurs\ in\ j-th\ clause}} \mathbf{4^j}$ and let $\mathbf{y} = \sum_{\mathbf{j=1,2,\ldots,m}} \mathbf{4^j}$. The $\mathbf{j}$-th digit of a possible sum is $\mathbf{1}$ in base 4 iff exactly one $\mathbf{x_i}$ is selected for the sum where $\mathbf{\hat{x}_i}$ occurs in the $\mathbf{j}$-th clause. Thus solvability of the given XSAT instance equals solvability of the new KNAPSACK instance.

Knapsack II

Remark 6.13: KNAPSACK can also be formulated as an optimisation problem. Given a list $x_1, x_2, \ldots, x_n$ and y, find the largest possible value the sum $\sum_{k=0,\ldots,n} x_k z_k$ can take without becoming strictly larger than y. Solving this problem is NP-hard. It is, however, unknown whether the problem is in NP. This is the case if and only if NP = CoNP.

Various optimisation problems are of that form, that solving them is as hard as NP-solving, but one cannot get the optimal number without accessing an NP-oracle (= a data base containing some NP-complete set) more than once. Note that such an oracle also provides negative evidence, that is, that something does not have a solution.

Function Graphs

A graph of a function f is in NP iff there is a nondeterministic polynomial time machine which, for all inputs x, has accepting computations outputting $f(x)$ but not outputting any other number in an accepting computation.

Example 6.14: Agrawal, Kayal and Saxena [2004] showed that one can check in polynomial time whether a number is prime. Thus one can also check whether a factorisation of a number (a list of numbers whose prodruct is that number) consists only of prime factors, that is, is the prime factorisation. The following two functions are in NP:

- Function computing the greatest prime factor;
- Function computing the number of prime factors.

NP algorithm guesses factorisation and verifies all prime; if so it computes output and accepts computation. These functions are believed neither to be in P nor to be NP-hard.

Homeworks 6.15-6.18

Homework 6.15: Find an algorithm which replaces a $\mathbf{n}$-variable $\mathbf{m}$-clause $\mathbf{k}$SAT formula by an equivalent $(\log(\mathbf{k})+1)$SAT formula with up to $\mathbf{n}+(\log(\mathbf{k})+1)\cdot\mathbf{m}$ variables and up to $\mathbf{m}\cdot\mathbf{k}$ clauses.

Homework 6.16: If one splits each $\mathbf{k}$-literal clauses into $\sqrt{\mathbf{k}}$ parts instead of $\mathbf{k}$ parts and starts with $\mathbf{n}$ variables and $\mathbf{m}$ clauses, how many new clauses and variables does one need in worst case? How is the value $\mathbf{k}$ updated?

Homework 6.17: What are the bounds for translating an $\mathbf{n}$-variable $\mathbf{m}$-clause $\mathbf{4}$SAT formula into a $\mathbf{3}$SAT formula? Why is a further translation into a $\mathbf{2}$SAT formula impossible?

Homework 6.18: Replace the proof sketch that 3OCCURSAT is NP-complete by a detailed proof that 3OCCUR3SAT is NP-complete, 3OCCUR3SAT consists of all satisfiable 3SAT formulas in which all variables occur at most thrice.

Homeworks 6.19-6.21

Homework 6.19: Prove that X3SAT is NP-complete. Here X3SAT is the set of all solvable XSAT instances where each clause has at most three literals. Show that NP-completeness also holds if one requires that all literals are positive, that is, for no variable x there is a literal of the form $\neg x$.

Homework 6.20: Prove that whenever A is in both NP and CoNP, then either NP = CoNP or the problem is not NP-complete.

Homework 6.21: Consider LOGKNAPSACK where all numbers x_k are in the set $\{0, 1, \ldots, n\}$. Is LOGKNAPSACK NP-complete or in P? Prove the answer.

Homeworks 6.22-6.26

Consider the following graph problems and prove that they are either NP-complete or in P.

Homework 6.22: Is there a roundpath visiting each node once (except that starting node = end node)?

Homework 6.23: Is there a roundpath using each edge exactly once (nodes might be visited several times)?

Homework 6.24: Can the nodes of the graph be coloured with two colours such that each neighbouring nodes have different colours?

Homework 6.25: Can the nodes of the graph be coloured with three colours such that each neighbouring nodes have different colours?

Homework 6.26: Can the nodes of the graph be coloured with four colours such that each neighbouring nodes have different colours?

Lecture 7

Lecture 7 will deepen the knowledge about the complexity of variants of SAT. As seen in Lecture 6, certain variants like 3SAT and XSAT allow for faster solutions of time $O(c^n)$ with $1 < c < 2$ than the usual method to test all combinations of variables which takes at least 2^n steps.

DPLL algorithms are named after Martin Davis, George Logeman, Donald Loveland and Hilary Putnam who, in two papers each with some of these authors laid the fundaments for these algorithms. They solve satisfiability problems where there are certain constraints on the clauses, for example, each variable occurring at most k times or each clause having at most k literals or the overall number m of clauses being bounded by kn. Under such additional constraints, the DPLL algorithms provide better bounds than the trivial algorithm to check all variables.

Example 7.1: DPLL Algorithm 3SAT

1. Algo(Clause Set $\mathbf{F}$, var $\mathbf{x_1, \ldots, x_n}$);
2. If there is an empty clause then return(false);
3. If there is no clause then return(true);
4. If there is a clause of form $\mathbf{y}$ with $\mathbf{y} \in \{\mathbf{x_k}, \neg\mathbf{x_k}\}$ then set $\mathbf{y} = \mathbf{1}$, remove all clauses containing $\mathbf{y}$ and remove $\neg\mathbf{y}$ from all remaining clauses and goto 2;
5. If there is a literal $\mathbf{y}$ such that $\neg\mathbf{y}$ occurs in no clause then remove all clauses containing $\mathbf{y}$ and goto 2;
6. Find variable $\mathbf{x_k}$ such that $($number of 2-literal clauses with $\mathbf{x_k}$, number of 3-literal clauses with $\mathbf{x_k}$$)$ is maximal;
7. If Algo$(\mathbf{F} \cup \{\mathbf{x_k}\}$,vars$)$ or Algo$(\mathbf{F} \cup \{\neg\mathbf{x_k}\}$,vars$)$ is true then return true else return false.

This is the basic algorithm. Better algorithms are known.

Rule Types

It is convenient to see DPLL algorithms as collections of rules in some order; always the first rule which applies is done.

Measure is the number of variables (or whatever other criterion is used to measure the complexity of input).

Simplification rules simplify the formula $\mathbf{F}$ by, for example, setting $\mathbf{y} = 1$ and doing the follow-up in the case that there is a single literal clause $\mathbf{y}$. The measure is not allowed to go up in simplification rules.

Branching rules are recursive calls which call several instances of the main program. Each branch leads to modifications of the formula which reduce the measure.

Branching Factors

If a branching rule has three branches (a), (b), (c) which reduce the measure by r_a, r_b, r_c respectively, then the branching factor is the least real number $s > 1$ such that

$$s^{-r_a} + s^{-r_b} + s^{-r_c} \leq 1.$$

Similarly for branchings with two, four or more outcomes.

Usually there are several different branching rules with different branching factors in the algorithm. One takes the maximum of these, call it $\tilde{s}$, and considers the initial measure, say n. An upper bound on the runtime of the algorithm is then $Poly(n) \cdot \tilde{s}^n$. Here note that $m \in O(n^3)$ for 3SAT; for general satisfiability formulas, this bound is not there and one would give the bound $Poly(n+m) \cdot \tilde{s}^n$.

Evaluating the 3SAT Algorithm

Each variable branched occurs both positive and negative. If it occurs in a $\mathbf{2}$-literal clause then this on one side of the branching disappears and on the other side becomes $\mathbf{1}$-literal clauses which fixes a further variable.

Use the following measure μ: If there is no $\mathbf{2}$-literal clause then $\mu = \mathbf{n}$ else $\mu = \mathbf{n} - \mathbf{0.21481}$.

If there is no $\mathbf{2}$-literal clause, then each branch of a variable will produce one and the measure decreases on both sides by $\mathbf{1.21481}$. Branching factor is $\mathbf{s}$ with $\mathbf{2s^{-1.21841} = 1}$, uprounded to $\mathbf{1.7693}$.

If there is a $\mathbf{2}$-literal clause, then this might disappear on both sides of the branching, but on one side of the branching a further variable will be removed. Branching factor is $\mathbf{s}$ with $\mathbf{s^{-1+0.21481} + s^{-2+0.21481} = 1}$, uprounded to $\mathbf{1.7693}$.

Modifying the 3SAT Algorithm

Example 7.2: Adding in a new simplification rule, called resolution: If there is a literal $\mathbf{y}$ such that $\mathbf{y}$ occurs in $\mathbf{3}$-literal clauses but $\neg\mathbf{y}$ only in $\mathbf{2}$-literal clauses, then add for all pairs of clauses $\alpha \vee \mathbf{y}$, $\beta \vee \neg\mathbf{y}$ the new clause $\alpha \vee \beta$ (it also has at most three literals) into $\mathbf{F}$ and afterwards remove all clauses containing $\mathbf{y}$ or $\neg\mathbf{y}$.

One uses only the number of current variables as measure and bills the measure gained by a resolution onto the next branching rule done; similarly with all other follow-up savings which are not taken into account.

Each branching rule except for the first and those immediately after a resolution will create one $\mathbf{2}$-literal clause; thus all branching rules except for the first will either eat up prior savings or remove on at least one side two variables. Branching factor is $\mathbf{s}$ with $\mathbf{s^{-1} + s^{-2} = 1}$, uprounded to $\mathbf{1.6181}$.

State of the Art

For $\mathbf{k}$SAT and deterministic algorithms, the following recent results are there: Work by Dantsin and coauthors 2002 (branching number $\mathbf{2k/(k+1)}$), Moser and Scheder 2011, Sixue Liu 2018.

Problem	2002	2011	2018
3SAT	1.5	1.3334	1.3280
4SAT	1.6	1.5001	1.4986
5SAT	1.6667	1.6001	1.5995
6SAT	1.7143	1.6667	1.6665

Improvements by continuous refinement of the algorithm (better case distinctions), usage of new ideas from coding theory (Uwe Schöning) combined with derandomisation; in other areas also the use of more sophisticated measures.

Example 7.3: An XSAT Algorithm

Exact satisfiability (XSAT) is a problem which has seen a lot of ongoing research and it is, after plain SAT and kSAT for constants $k \geq 3$ the most investigated variant of SAT in the field of exponential time algorithms.

Here the algorithms do not need a constraint on the number of literals per clause; the reason is that the number of satisfying assignments of a clause with k literals is at most k, as one literal has to be 1 and the other $k-1$ have to be 0; thus one gets k out of 2^k possible choices to consider when one branches the k variables of a k-literal clause. Note here that a subclause $x \vee x$ implies that $x = 0$ and a subclause $x \vee \neg x$ implies that all other literals are 0 while for x it does not matter whether it is 0 or 1; thus under the assumption that easy simplifications are done, any instance considered has for all literals of a clause different variables.

XSAT Algorithm – Simplification Rules

Always do the first of the below possibilities which applies.

- If $\mathbf{x}$ appears in a clause of form $\mathbf{x} \vee \mathbf{y}$, $\mathbf{x} \vee \mathbf{x} \vee \alpha$, $\mathbf{x} \vee \mathbf{y} \vee \neg\mathbf{y} \vee \beta$, $\mathbf{x}$ or both $\mathbf{x} \vee \mathbf{y} \vee \gamma, \mathbf{x} \vee \neg\mathbf{y} \vee \delta$ then set $\mathbf{x}$ to $\neg\mathbf{y}, \mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{0}$, respectively, and simplify accordingly.
- If a clause is of form $\mathbf{x} \vee \neg\mathbf{x}$ or $\mathbf{x} \vee \mathbf{y_1} \vee \ldots \vee \mathbf{y_k}$ with $\mathbf{y_1}, \ldots, \mathbf{y_k}$ not occurring elsewhere then remove it.
- If there are clauses α and $\alpha \vee \beta$ then set all literals in β to $\mathbf{0}$.
- If the instance is small and satisfiable then return $\mathbf{1}$.
- If a subinstance is small and unsatisfiable then return $\mathbf{0}$.
- If $\mathbf{x}$ appears both, positively and negatively, then pick clauses $\alpha \vee \mathbf{x}$, $\beta \vee \neg\mathbf{x}$ and replace all further clauses $\gamma \vee \mathbf{x}$ by $\beta \vee \gamma$ and $\delta \vee \neg\mathbf{x}$ by $\alpha \vee \delta$ and the two picked clauses by $\alpha \vee \beta$.

XSAT Algorithm – Branching Rules

- Branch the largest clause $\mathbf{x} \vee \mathbf{y} \vee \alpha$ with $\mathbf{x}, \mathbf{y}$ occurring elsewhere as $\mathbf{x} = \mathbf{0}, \mathbf{y} = \mathbf{0}$ versus $\mathbf{x} = \neg \mathbf{y}$ and simplify accordingly in both branchings.

Number of removed variables in dependence on number of literals in α.

Literals of α	case $\mathbf{x} = \mathbf{0}, \mathbf{y} = \mathbf{0}$	case $\mathbf{x} = \neg \mathbf{y}$
$\mathbf{1}$	$\mathbf{3}$	$\mathbf{3}$
$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$
$\mathbf{3}$ or more	$\mathbf{2}$	$\mathbf{5}$

The worst branching factor is $\mathbf{s}$ with $\mathbf{2s^{-3} = 1}$ and that satisfies $\mathbf{s < 1.2600}$. Gordon Hoi constructed in 2020 an $\mathbf{O(1.1674^n)}$ time algorithm.

7.4: The Exponential Time Hypotheses

All known algorithms for 3SAT, XSAT, X3SAT and other such problems use time $\Omega'(c^n)$ for some $c > 1$ and infinitely many n; this led to the following still unproven hypotheses.

The Exponential Time Hypothesis [Impagliazzo and Paturi 1999]: For each $k \geq 3$ there is a constant $c_k > 1$ such that for every correct kSAT algorithm there are infinitely many n for which some input with n variables needs at least c_k^n computation steps.

The Strong Exponential Time Hypothesis [Calabro, Impagliazzo and Paturi 2009]: The above c_k converge from below to 2.

Note that $2^n \cdot \mathrm{Poly}(m+n)$ is a trivial upper bound for all SAT problems including all kSAT.

Further Properties of ETH

Theorem 7.5 [Impaglazzio and Paturi 1999]: Assuming ETH with constants $c_3, c_4, \ldots$ as above, there is for each $k \geq 3$ a constant d_k such that, for all k and all kSAT algorithms, there are, for infinitely many n, instances of n variables and up to $n \cdot d_k$ clauses for which the kSAT algorithm needs at least c_k^n steps.

These constants d_k are only shown to exist; they are useful to prove that ETH, provided it holds, also forces other problems to use exponential time. This will be done on the next slides for various examples. The technique presented here is useful, if one has to prove conditional lower bounds under the assumption of ETH.

ETH applies also to XSAT Part I

Theorem 7.6: Assuming ETH, there are constants c_{xsat}, d_{xsat} such that every correct XSAT algorithm needs, for infinitely many n, on some instance with n variables and up to $d_{xsat} n$ clauses, at least time c_{xsat}^n computation steps to decide the solvability of this instance.

The idea is to translate a 3SAT instance with n variables and $d_3 n$ clauses. For each clause $x \vee y \vee z$ one introduces four new variables t, u, v, w only used for this purpose and replaces the clause by $x \vee v \vee w$ and $v \rightarrow y$ and $w \rightarrow z$; note that these two implications can be written with the XSAT clauses $v \vee \neg y \vee t$, $w \vee \neg z \vee u$. If v is 1, so is y; however, t is needed to make the clause satisfied when both $v, \neg y$ are 0.

ETH applies also to XSAT Part II

The so constructed instance has $\mathbf{n + 4d_3n}$ variables and up to $\mathbf{3d_3n}$ clauses.

Assume now that some algorithm solves this in time below $\mathbf{c_3^{n/(1+4d_3)}}$. Then this algorithm can be tweeked such that it also solves the input instances in time $\mathbf{c_3^n}$ by translating the input instance with $\mathbf{n}$ variables and up to $\mathbf{d_3n}$ clauses as above and then run the algorithm on this larger instance for $\mathbf{c_3^{(1+4d_3)n/(1+4d_3)} = c_3^n}$ not counting the translation time. This contradicts the choice of the input instances for 3SAT.

Thus the exponential time hypothesis also applies to XSAT. As all XSAT clauses constructed have only three literals, one has the following corollary.

Corollary: ETH applies also to X3SAT.

ETH with Respect to Clauses I

Theorem 7.7 [Impagliazzo and Paturi 1999]: The problem 3SAT parameterised by the number of clauses m obeys the Exponential Time Hypothesis, that is, if ETH holds for 3SAT with respect to n, then ETH also holds for 3SAT with respect to m.

Proof: By the ETH there are constants $c_3 > 1$, d_3 such that for every correct 3SAT algorithm there are infinitely many numbers n such that for each of these n there is an instance with n variables and at most $d_3 n$ clauses where the algorithm needs at least c_3^n computation steps. As $m \leq d_3 n$, the same algorithm needs, when measured with respect to m, at least $(c_3^{1/d_3})^m$ steps. Note that $c_3^{1/d_3} > 1$ as $c_3 > 1$ and $1/d_3$ is a positive rational number.

ETH with Respect to Clauses II

It remains to show that infinitely many different $\mathbf{n}$ imply infinitely many different $\mathbf{m}$.

For this one can use that difficult instances are reduced, that is, no simplification rule applies directly. Thus every variable appears at least twice and $\mathbf{m \geq 2/3n}$.

Now for each $\mathbf{n}$ in the above quantification there are only finitely many matching $\mathbf{m}$, as $\mathbf{2/3n \leq m \leq d_3n}$.

It follows that for every given correct algorithm, there are infinitely many $\mathbf{m}$ for which there is an instance with $\mathbf{m}$ clauses where the given algorithm uses at least $\mathbf{(c_3^{1/d_3})^m}$ steps. So ETH holds for 3SAT with parameter $\mathbf{m}$.

A Nontrivial Algorithm for SAT I

Theorem 7.8: There is a SAT solver using time $\mathbf{O(1.32472^m)}$ provided that every variable appears in a clause.

Simplification Rules:

- If there is a variable appearing only positive or only negative, then make all literals of this variable $\mathbf{1}$ and remove the corresponding clauses.
- If there is a variable with clauses $\neg\mathbf{x} \vee \alpha$, $\mathbf{x} \vee \beta_\mathbf{1}$, ..., $\mathbf{x} \vee \beta_\mathbf{h}$ then replace these clauses by $\alpha \vee \beta_\mathbf{1}, \ldots, \alpha \vee \beta_\mathbf{k}$ and remove variable $\mathbf{x}$; clause number goes down by one (Resolution Step 1).
- If there is a variable occurring as $\mathbf{x} \vee \alpha$, $\mathbf{x} \vee \beta$, $\neg\mathbf{x} \vee \gamma$, $\neg\mathbf{x} \vee \delta$ then replace these four clauses by $\alpha \vee \gamma$, $\alpha \vee \delta$, $\beta \vee \gamma$, $\beta \vee \delta$ and remove variable $\mathbf{x}$; clause number unchanged (Resolution Step 2).

A Nontrivial Algorithm for SAT II

Branching and Termination Rules:

- If there is no clause, return true; if there is an empty clause, return false.
- If none of the simplification and termination rules apply, then every variable appears at least thrice positively and twice negatively or vice versa. Branch one of the variables. Branching factor is s with $s^{-2} + s^{-3} = 1$ and $s < 1.32472$.

Note that simplification rules capture all variables which do not have at least five occurrences and for which at least two occurrences are positive and two occurrences are negative; thus the above branching rule is the only one needed and proves that the algorithm runs in time $O(1.32472^m)$. The state of the art is $O(1.2226^m)$ by Chu, Xiao, Zhang 2021.

SETH and Other Parameters of SAT

There is also concurrent research with respect to solving SAT with respect to the number ℓ of literals. Here the state of the art is $O(1.0641^{\ell})$ by Peng and Xiao [2021].

A SAT algorithm is of course also an algorithm for 3SAT, 4SAT, ...; thus the following corollary holds. Note that for this algorithm, the difficult cases are where there are less clauses than variables.

Corollary 7.9: SETH does not apply to kSAT with respect to the number of clauses or with respect to the number of literals.

Exponential Time - Polynomial Space

All the algorithms presented in this lecture use polynomial space. They are recursive calls where the depth of these calls is usually the number of variables or less, as each branching also removes one variable. Furthermore, the local parts – simplification rules and follow-up of branchings – can be done in polynomial time and do therefore only require polynomial sized local memory. Thus, at all stages of the algorithm and using that the various branches of a branching are tried out one after another, cause the overall algorithm only to use polynomial space.

Corollary 7.10 The algorithms of this lecture so far all use only polyomial space.

Exponential Time and Space

However, in some cases, exponential time algorithms also require exponential space, for example they create a exponential sized database. Data can be inserted in polynomial time and also presence of data in it can be checked, as usually the basic data base operations are in time logarithmic of the data base size.

An example is the algorithm “Meet in the Middle”, which branches half of the variables completely and writes all possible results into the database. Then the other half are branched equivalently and one checks for each outcome whether there is a matching entry in the data base. An example where this is done is if one wants to meet a specific sum like in the case of Knapsack.

Algorithm Meet in the Middle

ALgorithm 7.11 [Horowitz and Sahni 1974]: Consider a Knapsack Instance with n numbers $x_1, x_2, \ldots, x_n$ and target sum y.

In the first round, one runs over all binary $z_1, z_2, \ldots, z_{n/2}$ and computes

$$y_0 = \Sigma_{h=1,\ldots,n/2} z_h x_h$$

and writes each y_0 into the data base. In the second round, one runs over all binary $z_{n/2+1}, z_{n/2+2}, \ldots, z_n$ and computes

$$y_1 = \Sigma_{h=n/2+1,\ldots,n} z_h x_h$$

and checks for each y_1 whether $y - y_1$ is in the data base, that is, equal to some y_0. If there is a y_1 with matching y_0 then there is a solution else there is no solution.

The time complexity is $2^{n/2} \cdot \mathrm{Poly}(n)$.

Homeworks 7.12-7.15

For the following homeworks, improve the corresponding algorithms in the lecture with respect to computation time and verify your algorithm. It is not requested to meet the state of the art (those algorithms are very complicated).

Homework 7.12: Improve Algorithm 7.2 for 3SAT.

Homework 7.13: Provide an algorithm for 4SAT with nontrivial upper bound.

Homework 7.14: Improve Algorithm 7.3 for XSAT.

Homework 7.15: Improve the algorithm of Theorem 7.8 for SAT with respect to clauses.

Homeworks 7.16-7.18

Homework 7.16: Provide an algorithm in terms of number of literals ℓ with nontrivial upper bound ($O(1.4^{\ell})$ or better).

Assume for the following homeworks that ETH holds for SAT when parameterised by n or parameterised by m (choose what you need).

Homework 7.17: Prove that ETH holds for SAT when parameterised by ℓ.

Homework 7.18: Prove that ETH holds for SAT when parameterised by the number of clauses with at least five literals.

Homeworks 7.19-7.21

Homework 7.19: Prove that ETH holds for Knapsack (decision version) by reducing X3SAT.

Homework 7.20: Read up on data bases and explain in detail how the data base functions of the exponential size data base can be realised such that inserting data and checking of existance of data can both be done in polynomial time, that is, logarithmic in the size of the data base.

Homeworks 7.21: Provide an algorithm for XSAT based on the idea “meet in the middle”. Explain what is stored in the data base. This algorithm will not be competitive, neither in time nor space usage, but it is good to understand it.

Task 7.22

Task 7.22: Consider the following problem. Given a SAT instance with $\mathbf{n}$ variables and $\mathbf{n^3}$ clauses, does this instance have a satisfying assignment with at most $\mathbf{\log^3(n)}$ variables being $\mathbf{1}$?

Assuming the Exponential Time Hypothesis, prove that this problem is neither in $\mathbf{P}$ nor $\mathbf{NP}$-complete.

Note that the Exponential Time Hypothesis is needed for both separations.

Lecture 8

Lecture 8 will deal with topics related to counting. One is a promise problem: Given a SAT formula such that it either has 0 solutions or a number of solutions from some set A not containing 0. Is detecting a solution for this problem easier than solving SAT itself?

Furthermore, requiring an odd number of solutions (in place of just at least one solution) leads to the problem ParityP. How does this problem relate to NP?

Furthermore, one would like to count the number of solutions of a SAT formula, kSAT formula and so on. How difficult is this (perhaps in dependence of k)?

Random Polynomial Time

Definition 8.1: A problem $\mathbf{L}$ is in Random Polynomial Time (RP) if there is a probabilistic algorithm running in polynomial time such that for all words $\mathbf{w}$, if $\mathbf{w} \in \mathbf{L}$ then the algorithm says with probability at least $\mathbf{1/4}$ ACCEPT while for all words $\mathbf{w} \notin \mathbf{L}$ the algorithm always says REJECT.

Example 8.2: Given a multivariate polynomial $\mathbf{p}$ over the rational numbers with integer coefficients, the question whether this polynomial is nonzero on some inputs is in RP.

Algorithm: Let $\mathbf{n}$ be the number of variables and $\mathbf{d}$ be the degree of the polynomial. One draws at random for every variable a random number between $\mathbf{0}$ and $\mathbf{3d \cdot n}$ and one evaluates the polynomial at this number — if the result is $\mathbf{0}$ then the polynomial is everywhere $\mathbf{0}$ with probability at least $\mathbf{1/2}$; if the result is nonzero then the polynomial is not everywhere zero.

Verification of Algorithm I

A polynomial in one variable of degree d either has at most d zeroes or is everywhere zero. Let n be the number of variables. Assume that the overall polynomial is not zero everywhere.

If for one variable x there are more than d values where the polynomial when fixing x vanishes than the whole polynomial vanishes, as one has when fixing all variables besides x a single variable polynomial which vanishes at more than d places and thus the polynomial is zero everywhere. Thus when picking values for $x_1, x_2, \ldots, x_n$ at random and when the polynomial does not vanish, then the value chosen for each x_k hits with probability at most $1/3n$ a value where the polynomial, after $x_1, x_2, \ldots, x_k$ are fixed, vanishes.

Verification of Algorithm II

It follows that the overall probability to hit a zero of the polynomial is at most $1/3$ and so the procedure returns with probability $2/3$ a nonzero value of that if such a value exists.

A problem is that the polynomial might return a value up to $(3cnd)^d$ where c is the largest coefficient and one might be interested to have the dependence on the factor d reduced. This can be achieved by randomly choosing a prime p with $(3cn\log(d))^2 \leq p \leq (3cn\log(d))^3$ and then the computation of the polynomial value modulo p is polynomial in the number of digits of p and the degree d. This is in particular important if the formula of the polynomial is given by an arithmetic circuit and not an explicit list of all monomial terms with coefficients.

Remark: The problem whether a multivariate polynomial given by an arithmetic circuit has a nonzero value is the main known problem inside RP not known to be in P.

Theorem of Valiant and Vazirani I

Theorem 8.3 [Valiant and Vazirani 1986]: Assume that there is an RP-algorithm which outputs $\mathbf{1}$ with probability $\mathbf{1/2}$ if a $\mathbf{3SAT}$ formula ϕ has exactly one solution and outputs $\mathbf{0}$ always if ϕ has no solution. Then NP = RP.

Algorithm: The idea is to intersect the given problem with half spaces where each half space is given as randomly drawn set $\mathbf{W_k}$ of variables such that an assignment to $\mathbf{x_1, \ldots, x_n}$ is in the half space iff an odd number of the variables in $\mathbf{W_k}$ has the value $\mathbf{1}$ assigned. Now one draws uniformly half spaces $\mathbf{W_1, \ldots, W_n}$ and if the ϕ has a satisfying assignment then there is with probability $\mathbf{1/4}$ a $\mathbf{k}$ such that exactly one satisfying assignment is in $\mathbf{W_1 \cap \ldots \cap W_k}$. So one has $\mathbf{n}$ related problems to check.

Theorem of Valiant and Vazirani II

By assumption some RP algorithm $\mathbf{F}$ returns with probability $\mathbf{1/2}$ ACCEPT if there is exactly one assignment and always returns REJECT if there is no satisfying assignment.

Thus if ϕ is satisfiable then drawing random subspaces $\mathbf{W_1, \ldots, W_n}$ and checking each with $\mathbf{F}$ returns with probability $\mathbf{1/8}$ an ACCEPT in at least one of the tasks. If ϕ is not satisfiable, $\mathbf{F}$ would return also REJECT on all tasks $\phi \cap \mathbf{W_1} \cap \ldots \cap \mathbf{W_k}$.

For the borderline case that ϕ has one unique solution, note that this solution is in $\phi \cap \mathbf{W_1}$ with probability $\mathbf{1/2}$. This fact is taken into account for computing the overall probability $\mathbf{1/4}$ that one of the sets $\phi \cap \mathbf{W_1} \cap \ldots \cap \mathbf{W_k}$ has a solution. Furthermore, one has to do one additional test, namely whether the assignment of all variables to $\mathbf{0}$ is a solution, as that one would never go into the subspaces.

Theorem of Valiant and Vazirani III

The only problem is that $\phi \cap \mathbf{W_1} \cap \ldots \cap \mathbf{W_k}$ is not a 3SAT formula. Thus Valiant and Vazirani introduced for each $\mathbf{W_h}$ given by variables $\mathbf{x_{i_1}}, \ldots, \mathbf{x_{i_j}}$ new helper variables $\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_j}$ such that $\mathbf{y_1} \leftrightarrow \mathbf{x_{i_1}}$ and $\mathbf{y_{h+1}} \leftrightarrow \mathbf{y_h} \oplus \mathbf{x_{i_h}}$ and $\mathbf{y_j} \oplus \mathbf{x_{i_j}}$ is $\mathbf{1}$. Each clause in this formula can be translated into a 3SAT clause and so the intersection $\phi \cap \mathbf{W_1} \cap \ldots \cap \mathbf{W_k}$ becomes a formula with $\mathbf{n} + (\mathbf{nk})/\mathbf{2}$ variables expectedly, due to randomness, it can be slightly more or less, but at most $\mathbf{n}(\mathbf{k}+\mathbf{1})$ variables.

For example, the clause $\mathbf{y_1} \oplus \mathbf{x_2} \oplus \mathbf{y_2}$ would become the following four clauses: $\mathbf{y_1} \vee \mathbf{x_2} \vee \mathbf{y_2}$, $\mathbf{y_1} \vee \neg\mathbf{x_2} \vee \neg\mathbf{y_2}$, $\neg\mathbf{y_1} \vee \mathbf{x_2} \vee \neg\mathbf{y_2}$, $\neg\mathbf{y_1} \vee \neg\mathbf{x_2} \vee \mathbf{y_2}$. It are all clauses where an odd number of literals is positive.

Applications of the Theorem

One might ask to which degree can one strengthen the Theorem of Valiant and Vazirani. For this one formulates the Random Exponential Time Hypothesis: Every algorithm which evaluates satisfiable formulas with probability $1/8$ to ACCEPT and which always evaluates unsatisfiable formulas to REJECT runs on infinitely many instances in the worst case c^n time for some $c > 1$ where n is the number of variables.

Due to the quadratic increase in the number of variables, the Random Exponential Time Hypothesis implies, using the Theorem of Valiant and Vazirani, only this: Checking whether a 3SAT formula with at most one satisfying assignment is indeed satisfiable by a randomised algorithm requires, in the worst case, time $c^{\sqrt{n}}$ for some constant $c > 1$ independent of the algorithm.

Counting 2SAT Solutions I

Algorithm 8.4: Let ϕ be a 2SAT formula. Count(ϕ) does the first case which applies.

1. If formula ϕ consists of two disjoint parts with disjoint variable sets $\mathbf{X}$ and $\mathbf{Y}$, respectively, then return Count$(\phi_{\mathbf{X}}) \cdot$ Count$(\phi_{\mathbf{Y}})$.
2. If ϕ has $\mathbf{14}$ or less variables, solve them by brute-force and count the solutions in $\mathbf{O(1)}$ time.
3. If the formula consists of variables each having at most two neighbours then they either form a "line" or "circle" of $\mathbf{n}$ variables. Branch one or two of them such that they split into parts with $\mathbf{n/2 - 1/2}$ variables at most.
4. Branch the variable with the most neighbours (it are at least three).

Counting 2SAT Solutions II

Item one is a common item done in algorithms which exploits that there are two independent subproblems which can be solved by itself. The number of the overall solutions is the product of the number of solutions of each subproblem.

Item two is the bottom case which, due to $\mathbf{n \leq 14}$, counts only $\mathbf{O(1)}$.

Item three branches one or two variables. Then solving the halves has complexity $\mathbf{8 \cdot O(c^{n/2-1/2})}$ for the resulting halves in all four cases - they are different due to taking border conditions into account. As $\mathbf{n \geq 15}$, each item saves at least $\mathbf{8}$ variables, much better than $\mathbf{O(c^n)}$. The branching factor itself is in the worst case ($\mathbf{n = 15}$) the $\mathbf{s}$ with $\mathbf{8s^{-8} = 1}$ what is below $\mathbf{1.2969}$.

Counting 2SAT Solutions III

Item four has branching factor $\mathbf{s}$ with either $\mathbf{s^{-1}+s^{-4}=1}$ or $\mathbf{s^{-2}+s^{-3}=1}$, as each neighbour is forced to take a value in one of the two branchings.

For example, for a variable $\mathbf{x}$, if it occurs in clauses $\mathbf{x \vee u}$, $\mathbf{x \vee v}$ and $\mathbf{\neg x \vee w}$, one has to set $\mathbf{u, v = 1}$ in the case that $\mathbf{x = 0}$ and $\mathbf{w = 1}$ in the case that $\mathbf{x = 1}$. Thus one eliminates in one branch three and in the other branch two variables. Thus the branching factor is the $\mathbf{s}$ with $\mathbf{s^{-2}+s^{-3}=1}$, that is, $\mathbf{s = 1.32472}$. In the case that $\mathbf{x}$ occurs positively in all clauses (or negatively in all clauses), one eliminates only $\mathbf{x}$ in one branching and also all its neighbours in the other branching; hence the branching factor is the $\mathbf{s}$ with $\mathbf{s^{-1}+s^{-4}=1}$, that is, $\mathbf{s = 1.3803}$.

The overall branching factor is $\mathbf{\max\{1.2969, 1.32472, 1.3803\}}$ and the algorithm runs in time $\mathbf{O(1.3803^n)}$.

Unique Exponential Time Hypothesis

Definition 8.5: The Unique Exponential Time Hypothesis (UETH) says that there are constant c_3, d_3 such that for every algorithm there are infinitely many n and instances of 3SAT with n variables, up to $d_3 n$ clauses and either zero or one solutions such that the algorithm either makes a mistake or runs longer than c_3^n steps.

Theorem 8.6: Assuming UETH with c_3, d_3 as above, the runtime complexity to count 2SAT solutions is at least $\Omega'((c_3^{1/(d_3+1)})^n)$. In particular, for every correct algorithm counting the number of 2SAT solutions there are infinitely many n with instances of $(1 + d_3)n$ variables where the algorithm needs at least time $(c_3)^n$.

Construction of Theorem 8.6 I

Let a SAT instance ϕ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses be given, by assumption $\mathbf{m = d_3 n}$ and one can use clause repetition to get the exact bound. Now one takes $\mathbf{m}$ new variables $\mathbf{y_1, \ldots, y_m}$. For each literal $\mathbf{z}$ occurring in the $\mathbf{k}$-th clause (which is either of the form $\mathbf{x_h}$ or of the form $\neg\mathbf{x_h}$), one adds the implication $\mathbf{z \rightarrow y_k}$ into the 2SAT formula.

If an assignment to $\mathbf{x_1, \ldots, x_n}$ makes all clauses true, then all $\mathbf{y_1, \ldots, y_n}$ are forced to be $\mathbf{1}$. So this assignment contributes $\mathbf{1}$ to the number of all solutions of the 2SAT formula. If an assignment fails to make the $\mathbf{k}$-th clause true, then there is no condition on the value of $\mathbf{y_k}$ and therefore solutions to 2SAT based on that assignment can either have $\mathbf{y_k = 0}$ or $\mathbf{y_k = 1}$ without modifying any other variable. Thus there is an even number of solutions extending this assignment.

Construction of Theorem 8.6 II

For example, if the clauses are $\mathbf{x_1 \vee x_2 \vee x_3, \neg x_1 \vee \neg x_2 \vee x_4}$ then $\mathbf{y_1, y_2}$ are the new variables and the clauses are $\mathbf{x_1 \to y_1, x_2 \to y_1, x_3 \to y_1, \neg x_1 \to y_2, \neg x_2 \to y_2, x_4 \to y_2}$. Now $\mathbf{(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)}$ makes both clauses true and forces $\mathbf{y_1 = 1, y_2 = 1}$, so it contributes one 2SAT solution. However, $\mathbf{(x_1, x_2, x_3, x_4) = (0, 0, 0, 0)}$ makes only the second clause true and thus no condition implies $\mathbf{y_1 = 1}$, thus it contributes two 2SAT solutions.

As the instance ϕ in the proof has at most one solution, the overall number of 2SAT solutions is odd iff the instance ϕ is satisfiable. Thus one can compute the solvability of the 3SAT instance from the number of solutions of the new 2SAT instance. Furthermore, note that 3SAT is taken, as there the number of clauses is bounded by $\mathbf{d_3 n}$ what is not the case for the general SAT.

Related Results

Corollary 8.7: Given a 3SAT instance ϕ with $\mathbf{n}$ variables and $\mathbf{m}$ clauses, one can compute a 2SAT instance ψ with $\mathbf{n+m}$ variables and up to $\mathbf{3m}$ clauses such that ϕ has an odd number of solutions iff ψ has an odd number of solutions.

Corollary 8.8: If Parity3SAT requires exponential time, so does Parity2SAT. In particular Parity2SAT requires exponential time infinitely often under the assumption UETH.

NP-Hardness of Count2SAT

Corollary 8.9: Counting 2SAT instances is NP-hard.

Given a 3SAT instance ϕ with $\mathbf{n}$ variables and $\mathbf{m}$ clauses, one assigns to the $\mathbf{k}$-th clause a set $\mathbf{Y_k}$ of $\mathbf{n+1}$ new variables and for each variable $\mathbf{y} \in \mathbf{Y_k}$ and each literal $\mathbf{z}$ of the $\mathbf{k}$-th clause one puts the implication $\mathbf{z} \rightarrow \mathbf{y}$. Then there are for each assignment of ϕ with $\mathbf{h}$ unsatisfied clauses $(\mathbf{2^{n+1}})^{\mathbf{h}}$ many solutions of the new 2SAT instance ψ and thus the number of solutions of ϕ is that of ψ modulo $\mathbf{2^{n+1}}$; note that ϕ has at most $\mathbf{2^n}$ solutions.

Theorem 8.10: Counting 2SAT modulo a fixed number $\mathbf{k}$ is as hard as counting 3SAT modulo a fixed number $\mathbf{k}$, provided that one allows a polynomial increase of the number of variables.

Random kSAT

The following discussion is based on work of Akmal and Williams 2022. Consider the following promise problem.

Definition 8.11: R$_{\mathbf{q}}\mathbf{k}$SAT is the promise problem requesting an algorithm to REJECT in the case that there is no solution and to ACCEPT in the case that there are at least $\mathbf{q} \cdot \mathbf{2^n}$ solutions where $\mathbf{q}$ is a fixed positive rational and $\mathbf{k}$ a natural number.

Theorem 8.11: The problem R$_{\mathbf{q}}\mathbf{k}$SAT can be solved in polynomial time for all fixed $\mathbf{q}, \mathbf{k}$.

Note that the problem RP is not known to be in polynomial time; the main reason that this result does not transfer is that one uses the normal SAT and the normal SAT can have much longer and more clauses than $\mathbf{k}$SAT.

Proof of Theorem 8.11 I

A set $\mathbf{S}$ of clauses is called disjoint, if no two clauses have a variable in common. Note that a disjoint set of $\mathbf{h}$ $\mathbf{k}$SAT clauses has at most $\mathbf{2^{n-hk} \cdot (2^k - 1)^h}$ satisfying assignments. Thus the ratio of satisfying by all assignments is at most $\mathbf{((2^k - 1)/2^k)^h}$ what is below $\mathbf{q}$ for sufficiently high $\mathbf{h}$. That is, for every $\mathbf{q > 0}$ and $\mathbf{k \geq 1}$ there is a constant $\mathbf{h}$ where $\mathbf{((2^k - 1)/2^k)^h < q}$. Here $\mathbf{h}$ depends only on $\mathbf{k, q}$ and not on any further parameter of a given $\mathbf{k}$SAT instance.

The proof is done by induction. A polynomial time algorithm exists for 2SAT. So assume now that $\mathbf{F_k(\phi)}$ is $\mathbf{\{0, 1\}}$-valued where the value $\mathbf{1}$ is always taken when there are at least $\mathbf{q2^{n(\phi)}}$ solutions and $\mathbf{0}$ is always taken when there is no solution. Furthermore, assume that $\mathbf{F_k}$ runs in polynomial time. Now one uses $\mathbf{F_k}$ to construct $\mathbf{F_{k+1}}$.

Proof of Theorem 8.11 II

So let $\mathbf{h}$ be such that whenever there are $\mathbf{h}$ or more disjoint clauses then there are less then $\mathbf{q} \cdot \mathbf{2^n}$ solutions. Given an $(\mathbf{k}+\mathbf{1})$SAT instance ϕ, one first constructs with a greedy algorithm a maximal disjoint set – maximal in the sense that there is no disjoint superset of clauses. If it has $\mathbf{h}$ or more clauses one returns $\mathbf{0}$. Otherwise let $\mathbf{X}$ be the set of variables occurring in these clauses and consider, for each assignment $\mathbf{a}$ to these clauses, the formula $\phi_{\mathbf{a}}$ obtained by setting the variables in $\mathbf{X}$ according to the assignment $\mathbf{a}$. Let $\mathbf{n}(\phi_{\mathbf{a}})$ be the number of remaining variables. Now one returns $\mathbf{1}$ iff there is an assignment $\mathbf{a}$ to the variables in $\mathbf{X}$ for which $\mathbf{F}_{\mathbf{k}}(\phi_{\mathbf{a}})$ is $\mathbf{1}$.

Proof of Theorem 8.11 III

The correctness of $\mathbf{F_{k+1}}$ follows from that of $\mathbf{F_k}$ as now explained. Note that each clause has one literal with a variable in $\mathbf{X}$ and thus if ϕ is a $\mathbf{(k+1)}$SAT formula then all $\phi_\mathbf{a}$ are $\mathbf{k}$SAT formulas.

If ϕ has no solutions then so do all $\phi_\mathbf{a}$ and $\mathbf{F_k(\phi_a) = 0}$ for all assignments $\mathbf{a}$ to the variables in $\mathbf{X}$ and $\mathbf{F_{k+1}(\phi) = 0}$.

If ϕ has at least $\mathbf{q2^n}$ solutions, then at least one $\phi_\mathbf{a}$ must also have at least $\mathbf{q2^{n(\phi_a)}}$ solutions and $\mathbf{F_k(\phi_a) = 1}$ for this $\mathbf{a}$. Thus $\mathbf{F_{k+1}(\phi) = 1}$.

The values $\mathbf{k, h, q}$ are $\mathbf{O(1)}$ for fixed choices of these, as the size parameter is $\mathbf{n}$. Thus $\mathbf{F_{k+1}}$ consists of an $\mathbf{Poly(n)}$ search for the maximal disjoint set and, if this is below size $\mathbf{h}$, of $\mathbf{Poly(n)}$ handling for the variables to be set (they are constantly many, but the formula needs to be adjusted) and $\mathbf{O(1)}$ excecutions of $\mathbf{F_k(\phi_a)}$, thus $\mathbf{F_{k+1}}$ runs in $\mathbf{Poly(n)}$ time.

Results of Akmal and Williams

Theorem 8.13: Given a rational $q > 0$ and an integer k, there is a polynomial time algorithm to decide whether a given kSAT formula with n variables has at least $q \cdot 2^n$ solutions.

Theorem 8.14: Furthermore, there is a polynomial time algorithm which decides whether a 3SAT formula with n variables has at least $2^{n-1} + 1$ solutions.

Theorem 8.15: It is $\mathbf{NP}$-complete to decide whether a 4SAT formula with n variables has at least $2^{n-1} + 1$ solutions.

Classes BPP and PP

Definition 8.16: The class PP – Probabilistic Polynomial Time consists of all problems $\mathbf{L}$ for which a probabilistic polynomial time Turing machine accepts an input of size $\mathbf{n}$ with probability of at least $\mathbf{2^{-0.5} + 2^{-f(n)}}$ if it is in $\mathbf{L}$ and $\mathbf{f}$ is a fixed polynomial and rejects with probability at least $\mathbf{2^{-0.5}}$ if it is not in $\mathbf{L}$.

Definition 8.17: The class BPP – Bounded Probabilistic Polynomial Time consists of all problems $\mathbf{L}$ for which probabilistic Turing machine accepts with probability of at least $\mathbf{\frac{2}{3}}$ the members of $\mathbf{L}$ and rejects with probability at least $\mathbf{\frac{2}{3}}$ the nonmembers of $\mathbf{L}$. Here $\mathbf{\frac{2}{3}}$ can be replaced by any fixed threshold strictly greater than $\mathbf{\frac{1}{2}}$.

Homeworks 8.18-8.21

Prove that the following problems are hard under assumption of UETH with constants $\mathbf{c_3, d_3}$ to be used for the conditional lower runtime bounds below.

Homework 8.18: $\mathrm{Mod}_{\mathbf{3}}$2SAT, that is, whether the number of solutions of a 2SAT formula are nonzero modulo $\mathbf{3}$ with bound $\Omega'(\mathbf{c_3^{n/(1+2d_3)}})$.

Homework 8.19: $\mathrm{Mod}_{\mathbf{5}}$2SAT, that is, whether the number of solutions of a 2SAT formula are nonzero modulo $\mathbf{5}$ with bound $\Omega'(\mathbf{c_3^{n/(1+3d_3)}})$.

Homework 8.20: $\mathrm{Mod}_{\mathbf{7}}$2SAT, that is, whether the number of solutions of a 2SAT formula are nonzero modulo $\mathbf{7}$ with bound $\Omega'(\mathbf{c_3^{n/(1+4d_3)}})$.

Homework 8.21: Decision Knapsack with $\mathbf{n}$ variables and size of numbers being polynomial in $\mathbf{n}$ (the size is number of bits in binary representation).

Task 8.22 and Homeworks 8.23-8.24

Task 8.22: Show that determining whether a SAT formula has at least $2^{n-1}+1$ solutions is a problem in PP; note that this is not a promise problem but that the input should be rejected if the formula has at most 2^{n-1} solutions. Furthermore, show that under the assumption of UETH, this problem needs on infinitely many instances exponential time to be solved.

Homework 8.23: Prove that every problem in NP is also in PP.

Homework 8.24: A problem L is many-one reducible to RSAT via f, if f computes in polynomial time a satisfiability formula and for every $x \in L$, more than half of the assignments of $f(x)$ are solutions and for every $x \notin L$, $f(x)$ has no solution. Show that all problems many-one reducible to RSAT are in BPP and in RP.

Homework 8.25-8.27

Homework 8.25: Manders and Adleman [1976] showed that the following problem is NP-hard: Given three natural numbers $\mathbf{a, b, c}$ with input size $\mathbf{n}$, check whether there are natural numbers $\mathbf{x, y}$ with $\mathbf{ax^2 + by = c}$. Provide an exoponentuial time algorithm for the related problem to count all solutions $\mathbf{(x, y)}$ for this problem using time $\mathbf{Poly(n) \cdot 2^{n/2}}$.

Homework 8.26: Determine the complexity of the following related problem: Given natural numbers $\mathbf{a, b, c, d}$, count how many numbers $\mathbf{x}$ satisfy $\mathbf{ax^3 + bx^2 + cx = d}$.

Homework 8.27: Determine the complexity of the following related problem: Given natural numbers $\mathbf{a, b, c, d}$, count all tuples $\mathbf{(x, y, z)}$ such that $\mathbf{(a+2)^x + (b+2)^y + (c+2)^z = d}$. Under ETH, is the best algorithm in LOGSPACE, in P and requiring linear space or requiring exponential time?

Lecture 9

Some Problems like KNAPSACK of $\mathbf{m}$ numbers do not have known algorithms running in time $\mathbf{c^{m/2}}$ for any $\mathbf{c < 2}$, even when using exponential space. Another problem often coded is SAT itself. These problems are used to get conditional lower bounds near the trivial algorithms for three famous problems inside the class P. These three problems are 4SUM (Is the sum of four number in $\mathbf{S}$ equal to the target?), OVP (Are two binary vectors in $\mathbf{S}$ orthogonal, that is, disjoint), DVP (Are two ternary vectors in $\mathbf{S}$ different in all coordinates?). Here $\mathbf{S}$ is always a set of $\mathbf{n}$ objects of the type mentioned in the questions before.

A related problem where the lower bound does not follow from known others is 3SUM (Is the sum of three numbers in $\mathbf{S}$ equal to the target?). Here the problem itself became a benchmark problems to show other conditional lower bounds.

3SUM and 4SUM

Definition 9.1: For $k \geq 2$, the k-sum problem is the problem to decide whether a set of n numbers $x_1, x_2, \ldots, x_n$ contains k elements whose sum is a target number y; usually this target number is 0. For $n = 2$ the problem is in $O(n \log(n))$ and the reason is that one can use a database with logarithmic update and check time and first one writes all the n numbers into the data base and then one checks, for each number x_k whether $y - x_k$ is in the database.

This algorithm is quadratic for $k = 3$ and for $k = 4$, in the latter case, one would write all sums of two numbers into the data base and then check for all i, j whether $y - x_i - x_j$ is in the data base. So both have $O(n^2 \log(n))$ upper bound on their computational complexity.

Reducing KNAPSACK to 4SUM

Assume that the given Knapsack problem has $\mathbf{4m}$ numbers such that a subsum should give the target $\mathbf{z}$. Now group them in four groups of $\mathbf{m}$ numbers and let $\mathbf{n = 4 \cdot 2^m}$. Call the groups $\mathbf{V, W, X, Y}$ and let now $\mathbf{V', W', X', Y'}$ contain each all sums of some of the members of $\mathbf{V, W, X, Y}$, respectively, so each of $\mathbf{V', W', X', Y'}$ has up to $\mathbf{n/4}$ members. Now let $\mathbf{V'' = \{625v + 1 : v \in V'\}}$, $\mathbf{W'' = \{625w + 5 : w \in W'\}}$, $\mathbf{X'' = \{625x + 25 : x \in X'\}}$ and $\mathbf{Y'' = \{625y + 125 : y \in Y'\}}$. Furthermore, let $\mathbf{z'' = 625z + 156}$, note that $\mathbf{156 = 125 + 25 + 5 + 1}$. Thus four numbers $\mathbf{v'', w'', x'', y''}$ from $\mathbf{V'' \cup W'' \cup X'' \cup Y''}$ can only sum up to $\mathbf{z''}$ if $\mathbf{v'' \in V'', w'' \in W'', x'' \in X'', y'' \in Y''}$.

Now if an 4SUM algorithm uses only time $\mathbf{O(n^c)}$ with $\mathbf{c < 2}$ then the KNAPSACK algorithm for the original numbers uses only time $\mathbf{O(2^{cm}) = O((2^{c/4})^{4m})}$ where $\mathbf{2^{c/4} < \sqrt{2}}$.

3SUM and 4SUM

While the proof for 4SUM is based on KNAPSACK hardness, there is no such proof that 3SUM requires at least time $\mathbf{n^c}$ for all $\mathbf{c < 2}$. However, no algorithm which beats any $\mathbf{n^c}$ for any fixed $\mathbf{c < 2}$ for infinitely many $\mathbf{n}$ has been found, one conjectures it does not exist.

Statement 9.2: 3SUM Hypothesis: There is no $\mathbf{c}$ with $\mathbf{1 < c < 2}$ and no algorithm solving 3SUM correctly which runs in time $\mathbf{O(n^c)}$. Indeed, it is even conjectured that every algorithm solving 3SUM has for each $\mathbf{c < 2}$ the minimum complexity $\mathbf{n^c}$ for almost all $\mathbf{n}$.

The 3SUM Hypothesis is used in many applications to show conditional lower bounds; originally one wanted even to show that an algorithm needs $\mathbf{\Omega(n^2)}$ time, but this original version of the 3SUM conjecture was broken by Grønlund and Pettie 2014 who showed the upper bound of time $\mathbf{O(n^2 \cdot (\log\log n/\log n)^{2/3})}$ for 3SUM.

SUM, SUM', SUM* and Convolution

Definition 9.3: In each case, $\mathbf{n}$ numbers $\mathbf{x_1, x_2, \ldots, x_n}$ are given and in the first two cases also a target $\mathbf{y}$. Now an instance $(\mathbf{x_1, x_2, \ldots, x_n, y})$ is in the following set if the corresponding condition is satisfied:

3SUM: There are $\mathbf{i, j, k}$ with $\mathbf{x_i + x_j + x_k = y}$.

3SUM': There are $\mathbf{i, j, k}$ with $\mathbf{i \in X, j \in Y, k \in Z}$ and $\mathbf{x_i + y_j + z_k = y}$ where $\mathbf{X, Y, Z}$ are a partion of the possible indices (usually one splits the numbers into three equally sized sets).

3SUM*: There are $\mathbf{i, j, k}$ with $\mathbf{x_i + x_j = x_k}$.

3SUM*-Convolution: There are $\mathbf{i, j}$ with $\mathbf{x_i + x_j = x_{i+j}}$.

Theorem 9.4: All these problems are equivalent in the sense that if there is a $\mathbf{c < 2}$ such that one of them is solvable in $\mathbf{O(n^c)}$ time then this is true for all of them.

Sample Proof for SUM*

Recall that the 3SUM hypothesis says that 3SUM cannot be solved in time $O(n^c)$ for $c < 2$. Assume the 3SUM hypothesis holds. Then also 3SUM* cannot be solved in time $O(n^c)$ for any $c < 2$.

For this one takes a 3SUM instance with $y = 0$. Furthermore, one let r be a number at least 10 times as large as the largest of the numbers in the set. Now let, for $k = 1, \ldots, n$, the number $z_{2k-1} = r + x_k$ and $z_{2k} = 2r - x_k$. If $x_i + x_j + x_k = 0$ then $x_i + y_j = -x_k$ and $z_i + z_j = z_k$. If $z_{i'} + z_{j'} = z_{k'}$ then $z_{i'}, z_{j'} < 1.5r$ and $z_{k'} > 1.5r$, thus $z_{i'} = r + x_i$, $z_{j'} = r + x_j$ and $z_{k'} = 2r - z_k$ for some i, j, k; it follows that $x_i + x_j = -x_k$ and $x_i + x_j + x_k = 0$.

If now an algorithm needs time $O((2n)^c)$ for some $c < 2$ for 3SUM* then this time is also $O(n^c)$ and together with the linear time translation it decides 3SUM in time $O(n^c)$.

The Orthogonal Vector Problem

Definition 9.5: Two m-bit vectors are called orthogonal (or disjoint) if for every position e, at most one of them is 1. The Orthogonal Vector Problem (OVP) is the task to check whether a list of n m-bit vectors has two orthogonal vectors.

Definition 9.6: Two vectors are coordinate-wise distinct iff there is no coordinate on which both take the same value. The Distinct Vector Problem (DVP) is the task to check whether a list of n ternary m-digit vectors contains a pair of coordinatewise distinct vectors.

Definition 9.7: The Orthogonal Vector Hypothesis says that there are no $c < 2$ and no $d > 0$ such that some algorithm can solve OVP with n binary m-bit vectors in time $O(n^c m^d)$.

Definition 9.8: The Distinct Vector Hypothesis says that there are no $c < 2$ and no $d > 0$ such that some algorithm can solve DVP with n ternary m-bit vectors in time $O(n^c m^d)$.

The Problems OVP, DVP are SAT-hard

One starts with the following assumption.

Definition GSETH: This hypothesis says that there are no $\mathbf{c < 2}$ such that there is an algorithm which can decide SAT in time $\mathbf{O(c^n) \cdot Poly(m+n)}$. Note that here the polynomial in $\mathbf{n+m}$ can be chosen in dependence of the algorithm.

Note that GSETH is more general than SETH and SETH implies GSETH, but not vice versa. It is consistent with current knowledge that SETH might fail while GSETH still holds, the reason is that SETH only applies to $\mathbf{k}$SAT formulas.

Theorem 9.9: GSETH implies that the problems OVP and DVP cannot be solved in time $\mathbf{O(n^c m^d)}$ for any $\mathbf{c < 2}$ and $\mathbf{d > 0}$ with $\mathbf{n}$ being the number of vectors and $\mathbf{m}$ being the dimension (number of coordinates of the vectors).

Proof of Theorem 9.9 I

Assume that a SAT instance is given. One devides the $\mathbf{n}$ variables into two equal-sized lists $\mathbf{x_1, \ldots, x_{n/2}}$ and $\mathbf{x_{n/2+1}, \ldots, x_n}$, here $\mathbf{n/2}$ is down-rounded. Furthermore, $\mathbf{c_1, \ldots, c_m}$ is the list of clauses in this instance. One assumes that either OVH or DVH are false and uses the the corresponding problem is solvable in $\mathbf{(2^{n/2})^d \cdot Poly(m)}$ where one assumes in addition that $\mathbf{m \geq n}$, so that a polynomial in $\mathbf{n}$ could be replaced by one in $\mathbf{m}$ – note that SAT instances with less clauses than variables can be solved in time $\mathbf{O(1.2226^n)}$ as there is a $\mathbf{O(1.2226^m)}$ algorithm for SAT and $\mathbf{m \leq n}$; thus they do not need to be considered.

Now one creates sets $\mathbf{V, W}$ of $\mathbf{m}$-vectors which contain for each assignment of the first $\mathbf{n/2}$ and second $\mathbf{n/2}$ variables the vectors $\mathbf{v, w}$ which have at position $\mathbf{k}$ a $\mathbf{1}$ if the first respective second half of variables makes the clause true and a $\mathbf{0}$ otherwise.

Proof of Theorem 9.9 II

The next step depends on the problem to be shown to be hard.

Orthogonal Vector Problem: Let $\mathbf{U}$ to be the set of all vectors $\mathbf{(1 - u(1))(1 - u(2)) \ldots (1 - u(m))01 : u \in V}$ and all vectors $\mathbf{(1 - u(1))(1 - u(2)) \ldots (1 - u(m))10 : u \in W}$. Note that the 01 and 10 at the end enforce that one vector is taken from $\mathbf{V}$ and one from $\mathbf{W}$ when having an orthogonal pair. If these are derived from $\mathbf{v, w}$ then $\mathbf{v, w}$ cannot have at the same position a $\mathbf{0}$ and therefore the assignments used for $\mathbf{v, w}$ combine to an assignment for all variables which has for every clause $\mathbf{c_k}$ that either the first half or the second of of the variables are assigned such that $\mathbf{c_k}$ is true. A $\mathbf{(2^{n/2})^d \cdot Poly(m)}$ time algorithm for solving $\mathbf{U}$ produces then a $\mathbf{(2^{d/2})^n \cdot Poly(n + m)}$ time algorithm for SAT, in contradiction to GSETH; thus GSETH implies OVH.

Proof of Theorem 9.9 III

Now the other case.

Distinct Vector Problem: Let $\mathbf{U}$ to be the set of all vectors $\mathbf{v(1)v(2)\ldots v(m)1 : v \in V}$ and all vectors $\mathbf{(2\cdot w(1))(2\cdot w(2))\ldots(2\cdot w(m))2 : w \in W}$. Note that the 1 and 2 at the end enforce a pair of cordinate-wise distinct vectors in $\mathbf{U}$ must stem from vectors in $\mathbf{V}$ and $\mathbf{W}$, respectively. These cannot be $\mathbf{0}$ in both coordinate, but if they are $\mathbf{1}$ in both coordinates, the translated vectors will have $\mathbf{1}$ and $\mathbf{2}$, respectively. So again solving the coordinate-wise distinct vector problem in time $\mathbf{(2^{n/2})^d \cdot Poly(m)}$ gives on instance $\mathbf{U}$ a runtime of $\mathbf{(2^{d/2})^n \cdot Poly(n+m)}$ on the corresponding instance of SAT, in contradiction to GSETH. So GSETH also implies DVH.

The Binary Distinctness Case

Proposition 9.10: One can in time $\mathbf{O(mn\log(n))}$ check whether in a set $\mathbf{S}$ of $\mathbf{n}$ binary $\mathbf{m}$-bit vectors there are two vectors $\mathbf{x, y}$ which are binary distinct.

Proof: Note that two vectors $\mathbf{x, y}$ are coordinate-wise distince iff for all $\mathbf{k \in \{1, 2, \ldots, m\}}$ the condition $\mathbf{x(k) + y(k) = 1}$ holds, that is, $\mathbf{y}$ is the bit-complement of $\mathbf{x}$. Thus the idea is again a meet-in-the-middle algorithm: First one writes of each vector the complement into a data base. This uses up time $\mathbf{O(mn\log(n))}$, where the $\mathbf{m\log(n)}$ steps are needed to adjust the indices of the data base and to write the number into it. Then in a second phase, one goes again over the data base and checks for each vector $\mathbf{x}$, whether it is in the data base, that is, whether it is the bit-wise complement of another vector $\mathbf{y}$. This takes also time $\mathbf{O(mn\log(n))}$.

Homeworks 9.11-9.14

Complete the following items of the proof of Theorem 9.4. Assume that $\mathbf{c < 2}$.

Homework 9.11: Show that if 3SUM' can be solved in time $\mathbf{O(n^c)}$ then also 3SUM can be solved in time $\mathbf{O(n^c)}$.

Homework 9.12: Show that if 3SUM*-Convolution can be solved in time $\mathbf{O(n^c)}$ then also 3SUM can be solved in time $\mathbf{O(n^c)}$.

Homework 9.13: Show that if 3SUM can be solved in time $\mathbf{O(n^c)}$ so can 3SUM*-Convolution.

Homework 9.14: Provide an example of a set $\mathbf{X}$ such that four of its members sum to $\mathbf{0}$ and also two of its members sum to $\mathbf{0}$ but not three of its members sum to $\mathbf{0}$.

Homeworks 9.15 and 9.16

Homework 9.15: The numbers $\mathbf{243, 244, \ldots, 728}$ have all, when written as ternary numbers, six digits. View them as ternary six-digit vectors. Find the first $\mathbf{x > 243}$ such that the set of all numbers from $\mathbf{243}$ to $\mathbf{x}$ (inclusively of the borders) has two component-wise distinct six-digit numbers $\mathbf{y, z}$. Find these $\mathbf{y, z}$ and write them as both, ternary and decimal numbers.

Homework 9.16: The numbers $\mathbf{81, 82, \ldots, 242}$ have all, when written as ternary numbers, five digits. View them as ternary five-digit vectors. Find the largest $\mathbf{x < 242}$ such that the set of all numbers from $\mathbf{x}$ to $\mathbf{242}$ (inclusively of the borders) has two component-wise distinct five-digit numbers $\mathbf{y, z}$. Find these $\mathbf{y, z}$ and write them as both, ternary and decimal numbers.

Homeworks 9.17 and 9.18

Homework 9.17: Let $\mathbf{X} = \{\mathbf{00}, \mathbf{01}, \mathbf{02}, \ldots, \mathbf{98}, \mathbf{99}\}$ be the set of all two-digit numbers (when needed with leading zeroes). Find a subset $\mathbf{Y}$ as small as possible such that, whenever someone selects four different digits $\mathbf{i}, \mathbf{j}, \mathbf{k}, \mathbf{h}$ the set of all numbers in $\mathbf{Y}$ written only with these digits is nonempty and contains two component-wise distinct vectors.

Homework 9.18: Assume that

$$\mathbf{X_{m,n} = \{x_k = (k + m)^2 : k \in \{1, 2, \ldots, n\}\}}.$$

Find an example where $\mathbf{X_{m,n}}$ is in 3SUM* but not 3SUM*-Convolution. Furthermore, find a a function $\mathbf{f}$ such that $\mathbf{X_{m,f(m)}}$ is in SUM*-Convolution for all even $\mathbf{m \geq 2}$.

On the Midterm Test

In the second half of this lecture is the Midterm Test. The scope of the test is Lectures 1 – 7 and questions might relate to everything in these lectures. They will, however, be easier than the average homework, as for homeworks you can consult the internet and see what researchers did there. If branching factors or something like this are needed, they are listed out in the question paper.

This is a closed book exam and you should learn the material of the seven first lectures as good as possible. One A4 page helpsheet is allowed.

Lecture 10

Complexity of Concrete Problems in P. This lecture looks into P more from the perspective of complete problems. The lecture differs from Lecture 9 by mainly studying nontrivial upper bounds in cases where the lower bounds are either unknown (for matrix multiplication) or almost trivial (for multiplication).

Multiplication is, in contrast to addition, an operation where there is no trivial upper bound on the complexity - the bound $\mathbf{O(n)}$ for adding and subtracting $\mathbf{n}$-digit numbers is trivial as reading the input (or inspecting the input on the input tape) takes already that amount of time.

Furthermore, nondeterministic deciders are investigated which can use nondeterminism, but have in both cases (inside and outside the language) to provide a decision which is never wrong (but might be absent if wrong nondeterministic decisions are taken).

Analysis of Recurrences

Jon Bentley, Dorothea Blostein and James B. Saxe [1980] created a unified approach to resolve recurrence relations in complexity of algorithms with this theorem.

Theorem 10.1: Assume that an algorithm solves problems of size $\mathbf{O(nk)}$ by calling it self $\mathbf{h}$ times with subproblems of size $\mathbf{n}$ and furthermore produces local computations of time $\mathbf{O(n^\ell \cdot \log^p(n))}$. Then the following holds:

- If $\mathbf{\log(h)/\log(k) > \ell}$ then the time bound is $\mathbf{O(n^{\log(h)/\log(k)})}$;
- If $\mathbf{\log(h)/\log(k) = \ell}$ then the time bound is $\mathbf{O(n^{\log(h)/\log(k)} \cdot \log^{p+1}(n))}$.
- If $\mathbf{\log(h)/\log(k) < \ell}$ then the time bound is $\mathbf{O(n^\ell \log^p(n))}$.

Comments on Theorem 10.1

First note that for numbers which are not divisible by $\mathbf{k}$, one can make the input size a bit larger, for example adding zero-rows and columns for matrix multiplication or leading zeroes for integer multiplication.

Note that $\mathbf{p = 0}$ is possible; in this case, the second entry would have time bound $\mathbf{O(n^\ell \log(n))}$ and so a logarithmic factor comes in.

For $\mathbf{\log(h)/\log(k)}$ to be a rational number, one needs that $\mathbf{h = k^q}$ for some positive rational number $\mathbf{q}$. Usually irrational numbers are uprounded at the fourth or fifth digit after the decimal dot. So if a problem is solvable in $\mathbf{O(n^{\log(3)})}$ then it is also solvable in time $\mathbf{O(n^{1.58497})}$ due to uprounding.

Strassen's Algorithm I

Another complexity value to which often is referred is the amount of time needed to multiply two $\mathbf{n \times n}$ matrices $\mathbf{A, B}$. The standard algorithm computes $\mathbf{C = A \times B}$ by letting $\mathbf{C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}}$. This algorithm uses time $\mathbf{O(n^3)}$, as it does $\mathbf{n}$ multiplications and additions of matrix elements to compute each $\mathbf{C_{i,j}}$ and there are $\mathbf{n^2}$ values to be computed.

Theorem 10.2 [Strassen 1969] Two $\mathbf{n \times n}$ matrices can be multiplied with $\mathbf{O(n^{\log(7)}) = O(n^{2.8074})}$ basic operations.

For this, one divides $\mathbf{A, B, C}$ into two $\mathbf{m \times m}$ blocks with $\mathbf{m = n/2}$ or $\mathbf{m = n/2 + 1/2}$, the number $\mathbf{m}$ is an integer and in the second case one adds a $\mathbf{ZERO}$-row and -column to $\mathbf{A, B}$. Strassen schowed that then the product of $\mathbf{A \times B}$ is a linear combination of $\mathbf{7}$ matrix multiplications of matrices of size $\mathbf{m \times m}$ and one has $\mathbf{Time(n) = 7 \cdot Time(n/2) + O(n^2)}$ instead of $\mathbf{Time(n) = 8 \cdot Time(n/2)}$. This recurrrence gives the time bound $\mathbf{O(n^{\log(7)})}$.

Strassen's Algorithm II

Split $\mathbf{A}, \mathbf{B}, \mathbf{C}$ into $\mathbf{A_{1,1}}, \mathbf{A_{1,2}}, \mathbf{A_{2,1}}, \mathbf{A_{2,2}}$, $\mathbf{B_{1,1}}, \mathbf{B_{1,2}}, \mathbf{B_{2,1}}, \mathbf{B_{2,2}}$, $\mathbf{C_{1,1}}, \mathbf{C_{1,2}}, \mathbf{C_{2,1}}, \mathbf{C_{2,2}}$ and now let

$$\begin{aligned}
\mathbf{D_1} &= (\mathbf{A_{1,1}} + \mathbf{A_{2,2}}) \times (\mathbf{B_{1,1}} + \mathbf{B_{2,2}}), \\
\mathbf{D_2} &= (\mathbf{A_{1,1}} + \mathbf{A_{2,2}}) \times (\mathbf{B_{1,1}}), \\
\mathbf{D_3} &= (\mathbf{A_{1,1}}) \times (\mathbf{B_{1,2}} - \mathbf{B_{2,2}}), \\
\mathbf{D_4} &= (\mathbf{A_{2,2}}) \times (\mathbf{B_{2,1}} - \mathbf{B_{2,2}}), \\
\mathbf{D_5} &= (\mathbf{A_{1,1}} + \mathbf{A_{1,2}}) \times (\mathbf{B_{2,2}}), \\
\mathbf{D_6} &= (\mathbf{A_{2,1}} - \mathbf{A_{1,1}}) \times (\mathbf{B_{1,1}} + \mathbf{B_{1,2}}), \\
\mathbf{D_7} &= (\mathbf{A_{1,2}} - \mathbf{A_{2,2}}) \times (\mathbf{B_{2,1}} + \mathbf{B_{2,2}}), \\
\mathbf{C_{1,1}} &= \mathbf{D_1} + \mathbf{D_4} - \mathbf{D_5} + \mathbf{D_7}, \\
\mathbf{C_{1,2}} &= \mathbf{D_3} + \mathbf{D_5}, \\
\mathbf{C_{2,1}} &= \mathbf{D_2} + \mathbf{D_4}, \\
\mathbf{C_{2,2}} &= \mathbf{D_1} - \mathbf{D_2} + \mathbf{D_3} + \mathbf{D_6}.
\end{aligned}$$

Multiplying in $\mathbf{O((n/2)^{2.8074})}$; adding, subtracting in $\mathbf{O((n/2)^2)}$.

Strassen's Algorithm III

The ideas of Strassen's algorithm were brought a further step forward by subsequent work and asymptopically better matrix multiplications were found. Currently, Alman and Willians hold the record with matrix multiplication in time $\mathbf{O(n^{2.37286})}$ for multiplying two $\mathbf{n \times n}$ matrices; furthermore, for certain other rings than rationals or integers like the ring of a finite field, even slightly better bounds are known.

Valiant [1975] showed that matrix multiplication can be used to check membership of words in a context-free grammar and the current bound for this problem is therefore also $\mathbf{O(n^{2.37286})}$. On the other hand, Lee [2002] showed that if one can recognise context-free grammars in time $\mathbf{O(n^c)}$ then one can do matrix multiplication in time $\mathbf{O(n^{3-(3-c)/3})}$. However, there is still a gap between the two bound-translations and therefore it is an open problem whether the two complexities coincide.

The Karatsuba Algorithm

Theorem 10.3 [Karatsuba 1960]. Multiplication of two $\mathbf{n}$ bit integers can be done in time $\mathbf{O(n^{\log(3)}) = O(n^{1.58497})}$.

Given $\mathbf{2n}$ bit integers $\mathbf{x + 2^n y}$ and $\mathbf{v + 2^n w}$ with $\mathbf{x, y, v, w}$ all having $\mathbf{n}$ bits, it holds that

$$\mathbf{(x+2^n y)\cdot(v+2^n w) = xv+2^n((x+y)(v+w)-xv-yw)+4^n yw}$$

and one needs, beyond shifting of digits, three multiplications $\mathbf{vx, yw, (x+y)(v+w)}$ of $\mathbf{(n+1)}$-bit integers. Thus $\mathbf{F(2n) = 3F(n+1) + O(n)}$ operations. Ignoring the one bit length-increase by forming $\mathbf{x+y, v+w}$, one gets that the overall algorithm is $\mathbf{O(n^{\log(3)})}$. With this result, Anatoly Karatsuba disproved the conjecture mentioned by Andrey Kolmogorov that multiplication requires $\mathbf{\Omega(n^2)}$ time.

Example 10.4

One can also split an input three-way as follows:

$$(x+2^n y+4^n z)\cdot(u+2^n v+4^n w) = a+2^n b+4^n c+8^n d+16^n e,$$

where $a = x\cdot u,$

$$e = z\cdot w,$$
$$b = (x+y)\cdot(u+v) - a - y\cdot v,$$
$$d = (y+z)\cdot(v+w) - e - y\cdot v,$$
$$c = (x+y+z)\cdot(u+v+w) - b - d + y\cdot v.$$

This splitting is an easy method and it gives $F(3n) = 6\cdot F(n)$, as there are six multiplications, one for each of a, e, three more for b, d which are $(x+y)\cdot(u+v)$, $(y+z)\cdot(v+w)$, $y\cdot v$ plus the usage of a, e computed before. The last value c uses $(x+y+z)\cdot(u+v+w)$ and $b, d, y\cdot v$ all computed before. This gives $O(n^{\log(6)/\log(3)})$ what is in $O(n^{1.63093})$ and worse than Karatsuba's halving method.

Improvements by Andrei Toom

Multiplication with constants (independent of $\mathbf{n}$) is $\mathbf{O(n)}$. Note that the result of the multiplication is the value of a polynomial at input $\mathbf{2^n}$, where the unknowns $\mathbf{a, b, c, d, e}$ are the coefficients of the polynomial. One can compute this polynomial at $\mathbf{i = -2, -1, 0, 1, 2}$ by evaluating $\mathbf{p(i) = (x + iy + i^2z) \cdot (u + iv + i^2w)}$ for these five values of $\mathbf{i}$; so one has five multiplications of numbers with up to $\mathbf{n + 3}$ digits, the constant $\mathbf{3}$ here is ignored.

Now let $\mathbf{V}$ be the Vandermonde matrix with $\mathbf{V_{i,j} = i^j}$ where $\mathbf{j = 0, 1, 2, 3, 4}$ (the indices of entries in $\mathbf{V}$ are a bit input driven and nonstandard). Here $\mathbf{0^0 = 1}$. As the determinant of $\mathbf{V}$ is the product of all differences between two distinct values of $\mathbf{i}$ and thus nonzero, $\mathbf{V}$ has an inverse $\mathbf{W}$ which can be precomputed (it is independent of $\mathbf{n}$ and all inputs) and $\mathbf{(a, b, c, d, e) = W \cdot (p(-2), p(-1), p(0), p(1), p(2))}$. Thus one can do with five small multiplications.

The Multiplication of Toom

Theorem 10.5: One can mulitply two $\mathbf{n}$-bit numbers in time $\mathbf{O(n^{\log(2k-1)/\log(k)})}$ and this can be brought as close to $\mathbf{O(n)}$ as desired by choosing a big $\mathbf{k}$; note that $\mathbf{\log(2k) = \log(k) + 1}$, thus $\mathbf{\log(2k-1)/\log(k) < 1 + 1/\log(k)}$ and converges to $\mathbf{1}$ for $\mathbf{k}$ going to infinity.

Preliminaries: Multiplying with fixed rational constants, with multiplying with $\mathbf{2^{hn}}$ for small $\mathbf{h}$ (bit shifting) and adding $\mathbf{n}$-bit numbers are all in $\mathbf{O(n)}$. Similarly the multiplication with a fixed size matrix with fixed rational coefficients with a constant dimension vector is $\mathbf{O(n)}$.

Toom's Algorithm is a family of algorithms and once one has fixed $\mathbf{k}$, one considers it to be constant. The larger the numbers to be handled are, the better is a big $\mathbf{k}$. So from now on let $\mathbf{k}$ be fixed and $\mathbf{k \geq 2}$.

Toom's Algorithm for Fixed $\mathbf{k}$

Algorithm 10.6: One precomputes the Vandermonde matrix with entries $\mathbf{i^j}$ for $\mathbf{i = -(k-1), \ldots, (k-1)}$ and $\mathbf{j = 0, 1, \ldots, 2k-2}$ and also computes the inverse $\mathbf{W}$ which translates the $\mathbf{2k-1}$-vector $\mathbf{(p(1-k), p(2-k), \ldots, p(k-2), p(k-1))}$ into $\mathbf{z_0, z_1, \ldots, z_{2k-2}}$ so that the result of the planned multiplication is the sum over all $\mathbf{2^{jn} \cdot z_j}$. The entries of $\mathbf{W}$ are fixed rational numbers; here $\mathbf{i^0 = 1}$ for all $\mathbf{i}$ including $\mathbf{i = 0}$.

For multiplying two $\mathbf{kn}$-bit numbers, one splits them into $\mathbf{k}$ blocks $\mathbf{x_0, x_1, \ldots, x_{k-1}}$ and $\mathbf{y_0, y_1, \ldots, y_{k-1}}$ of length $\mathbf{n}$ and computes $\mathbf{p(i) = (\sum_j i^j \cdot x_j) \cdot (\sum_h i^h \cdot y_h)}$. Then one multiplies the result with $\mathbf{W}$ and obtains $\mathbf{z_0, z_1, \ldots, z_{2k-1}}$. One outputs $\mathbf{p(2^n) = \sum_j (2^n)^j \cdot z_j}$.

Performance and Follow-Ups

Additions and multiplications with fixed rational constants independent of $\mathbf{n}$ and inputs are $\mathbf{O(n)}$ and additions and multiplying with $\mathbf{2^n}$ (shifting of bits). Thus the complexity depends on doing the $\mathbf{2k-1}$ small multiplications, as those need time $\mathbf{\Omega(n)}$. So the time complexity of Toom's Algorithm satisfies the recurrence $\mathbf{F(kn) = (2k-1) \cdot F(n) + O(n)}$ and this gives time $\mathbf{O(n^{\log(2k-1)/\log(k)})}$ for Toom's algorithm with parameter $\mathbf{k}$.

Arnold Schönhage and Volker Strassen [1971] constructed an integer multiplication algorithm running in time $\mathbf{O(n \cdot \log(n) \cdot \log\log(n))}$. Manfred Fürer [2007] improved this bound to $\mathbf{n \cdot \log(n) \cdot 2^{O(\log^*(n))}}$ where $\mathbf{\log^*(n) = \min\{h \geq 1 : \log^{(h)}(n) \leq 1\}}$.

The state of the art for the complexity of multiplication is $\mathbf{O(n \cdot \log(n))}$ by David Harvey and Joris van der Hoeven [2021].

Nondetermistic Deciders I

A nondeterministic polynomial time decider outputs on $x \in L$ on some computation ACCEPT and on $x \notin L$ on some computation REJECT; the machine is, however, good, that is always run within some polynomial time bound and never outputs the wrong decision, while it might abstain on many runs.

The nondeterministic versions NETH and NSETH of ETH and SETH postulate lower bounds for nondeterministic deciders.

Definition 10.7: The hypothesis NETH says that there is are $c, d > 1$ such that every nondeterministic decider for 3SAT instances needs at least time c^n for infinitely many instances where the number of clauses is bounded by $d \cdot n$ for the number n of variables.

Nondeterministic Deciders II

Definition 10.8: The hypothesis NSETH by Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi and Stefan Schneider [2016] says that there is are no constants $\mathbf{c < 2, d > 0}$ such that one can solve all SAT instances in nondeterministic time $\mathbf{O(c^n \cdot (n + m)^d)}$.

Theorem 10.9 [Carmosino, Gao, Impagliazzo, Mihajlin, Paturi and Schneider 2016]: For any $\mathbf{c > 1.5}$, 3SUM with numbers bounded by $\mathbf{s(n)}$ for some polynomial $\mathbf{s}$ has a nondeterministic decider running in time $\mathbf{O(n^c)}$.

Theorem 10.10: If 3SUM has a decider in $\mathbf{NTIME(n^c)}$ with $\mathbf{c < 1.5}$ then KNAPSACK has a decider in $\mathbf{NTIME(2^{(c/3)n})}$.

Proof of Theorem 10.9 I

First note that for nondeterministic deciders, the difficulty is to make sure that there is no solution when rejecting the input, that is, saying “NO SOLUTION”. For the positive side, the nondeterministic decider has only to guess the indices and to add up the corresponding numbers, what is $\mathbf{O(n)}$. So assume there is no solution, the following algorithm explains how to verify this. Let $\mathbf{Q}$ be the set of the $\mathbf{n}$ input numbers and $\mathbf{q}$ the target output (usually $\mathbf{0}$). Let $\mathbf{d = (c + 3/2)/2}$.

Let $\mathbf{P}$ be the set of prime numbers between $\mathbf{n^d}$ and $\mathbf{n^d \cdot e}$, there are approximately $\mathbf{n^d/\log(n)}$ such primes, the constant $\mathbf{e}$ has to be chosen appropriately for this. Choose that $\mathbf{p \in P}$ such that modulo $\mathbf{p}$ there are the least number of triples $\mathbf{(x, y, z)}$ with $\mathbf{x + y + z = q}$.

Proof of Theorem 10.9 II

For this, one uses the data base approach to make a polynomial $r(u)$ of degree p such that each coefficient a_k of u^k is the number of numbers in Q which are modulo p equal to k.

Polynomial multiplication satisfies, similar to Toom's algorithm, that one can compute $r(u)^3$ in time $O(n^{d'})$ for each given $d' > c$, in particular for some $d' < c$. For this note that the coefficients are smaller than n and thus, in all computations, the number of binary digits of the coefficients is always logarithmic, so that handling these numbers gives only a polylogarithmic overhead.

The overall number of solutions modulo p is in $n^3/|P|$ and by choice of the constant e, this is smaller than $n^{3/2}$, as $|P| \geq n^d/\log(n)$ and thus is greater than $n^{3/2}$.

Proof of Theorem 10.9 III

One had chosen that the size of the numbers is bounded by $\mathbf{s(n)}$, thus two distinct numbers $\mathbf{v, w}$ between $\mathbf{-3s(n)}$ and $\mathbf{3s(n)}$ can coincide only modulo $\mathbf{\log(6s(n)/\log(n^d)) + 2}$ many primes in $\mathbf{P}$ what is $\mathbf{O(1)}$. Thus the average number of solutions modulo a random prime in $\mathbf{P}$ is

$$\mathbf{(\log(6s(n)/(n^d)) + 2) \cdot (n^3 \cdot \log(n)/(n^d))}$$

and the exact number is the sum of the coefficients of the polynomial the polynomial $\mathbf{r(u)}$ at $\mathbf{0, p, 2p, 3p}$; this number is below above average given in the displayed formula. One guesses these triples $\mathbf{(x, y, z)}$ in ascending lexicographic order and verifies that (a) $\mathbf{x + y + z = q}$ modulo $\mathbf{p}$ but (b) $\mathbf{x + y + z \neq q}$ (without any modulo). If this goes through, one goes to REJECT.

Discussions of Thms 10.9 & 10.10

Theorem 10.10 is the counterpart of the corresponding result showing the 4SUM hardness provided that KNAPSACK cannot be solved in deterministic time $\mathbf{O(c^n)}$ with $\mathbf{c < \sqrt{2}}$. However, this assumption is quite strong; the next result, Theorem 10.11, will have a weaker assumption and therefore also show less.

Indeed, Carmosino, Gao, Impagliazzo, Mihajlin, Paturi and Scheiner [2016] used their 3SUM result to show that, assuming NSETH, one cannot reduce problems like general SAT to show that 3SUM cannot be solved in $\mathbf{O(n^c)}$ for $\mathbf{c < 2}$. Such a reduction would then also be usable to show that a nondeterministic decider for 3SUM translates into a nontrivial upper bound for nondeterministic deciders of SAT which would disprove NSETH. Thus assuming NSETH, one cannot construct an almost quadratic lower bound for 3SUM using the difficulty of SAT.

Consequences of NETH I

Theorem 10.11: Assume that NETH holds for parameter ℓ (number of literals). Then there is no $c > 0$ such that for all k the problem kSUM are in $\mathrm{NTIME}(n^c)$, that is, kSUM has no uniform nondeterministic polynomial time decider dealing with all k.

Proof: One considers solving SAT instance with ℓ literals and let $d > 1$ be a constant such that one cannot solve all SAT instances in time d^ℓ.

Now one adds to each clause C_i with j_i literals $j_i - 1$ new variables only occurring there and requires that for a solution that the number of satisfied literal for the i-th clause is exactly j_i. The overall number of literals is now bounded by 2ℓ for the new instance. The number of variables is also bounded by 2ℓ.

Consequences of NETH II

Solving the new problem is equivalent to solving the old one, as the new version of clause $\mathbf{C_i}$ can have $\mathbf{j_i}$ literals true exactly when the old one has at least one literal true – one makes of the new variables exactly so many true as one needs to reach $\mathbf{j_i}$.

By assumption the new problem cannot be solved in time $\sqrt{\mathbf{d}}^{\mathbf{2}\ell}$. This new problem is, however, an XSAT style problem, as every clause is true only when the number of satisfying literals is an exact value depending on a clause.

Now one selects $\mathbf{c}, \mathbf{k}$ such that there is a $\mathbf{k}$SUM algorithm running in time $\mathbf{O(n^c)}$ and $(\mathbf{2^{2\ell/k}})^\mathbf{c} < \mathbf{d}^{\ell/4}$; after taking logarithms one gets $\mathbf{2\ell/k \cdot c < \log(d)\ell/4}$ and by isolating $\mathbf{k}$ on the right side one gets $\mathbf{8c/\log(d) < k}$.

Consequences of NETH III

Now one has to code this into a KNAPSACK problem. For this, one distributes the variables onto $\mathbf{k}$ pools with each having at most $\mathbf{2\ell/k}$ variables. Furthermore, for each clause having $\mathbf{2h-1}$ literals, one reserves a block of $\mathbf{h}$ bits in a binary representation of numbers coding the solvability of the modified SAT instance. As the original clause has $\mathbf{h}$ members and the sum of all clause-lengths is ℓ, it follows that the so constructed binary number has ℓ bits and that the number of the true literals for each clause coded with $\mathbf{h}$ bits is in any assignment between $\mathbf{0}$ and $\mathbf{2h-1}$. Note that $\mathbf{2h-1 < 2^h}$ for all $\mathbf{h}$, this can be verified by checking the initial values: $\mathbf{2 \cdot 1 - 1 < 2^1}$, $\mathbf{2 \cdot 2 - 1 < 2^2}$, $\mathbf{2 \cdot 3 - 1 < 2^3}$, $\mathbf{2 \cdot 4 - 1 < 2^4}$ and so on.

Thus one gets a KNAPSACK instance (as on Slide 209) and here the size of the numbers is $\mathbf{2^\ell}$ what is $\mathbf{(2^{2\ell/k})^{k/2}}$.

Consequences of NETH IV

In this KNAPSACK instance, one evaluates for every block of $\mathbf{2\ell/k}$ numbers the corresponding assignments and then puts down for each clause of $\mathbf{2h-1}$ literals in the corresponding $\mathbf{h}$-bit slot the number of satisfied literals. This gives $\mathbf{2^{2\ell/k}}$ many numbers for each set.

Thus for the given choice of $\mathbf{k}$, when one distributes the input of $\mathbf{n = 2^{2\ell/k} \cdot k}$ numbers on the corresponding $\mathbf{k}$ subsets, then one gets an instance where each number is binary and has ℓ bits and the problem whether one can find $\mathbf{k}$ numbers, one from each set, such that there sum is equal to $\mathbf{q}$ where $\mathbf{q}$ is the natural number obtained by setting the value for each clause of of size $\mathbf{2h-1}$ to $\mathbf{h}$ in its block of $\mathbf{h}$ bits, cannot be solved by a nondeterministic decider in nondeterministic time $\mathbf{O(n^c)}$. Note that the numbers are bounded by $\mathbf{n^k}$, thus for the specific counterexample to the bound $\mathbf{c}$, the polynomial $\mathbf{s(n)}$ used in Theorem 10.9 exists.

Homework 10.12-10.14

Homework 10.12: Show that polynomials in one variable over a ring can be multiplied with $\mathbf{O(n^{\log(3)})}$ ring operations, where $\mathbf{n}$ is the maximum degree of the polynomials multiplied.

Homework 10.13: Assume that one has numbers $\mathbf{x + \sqrt{d}y}$ and $\mathbf{v + \sqrt{d}w}$ represented as pairs $\mathbf{(x, y), (v, w)}$ of integers where $\mathbf{d}$ is a small integer constant which is not a square and it can be considered to be constant. How many integer multiplications are needed to compute $\mathbf{(p, q)}$ with $\mathbf{p + \sqrt{d}q = (x + \sqrt{d}y) \cdot (v + \sqrt{d}w)}$?

Homework 10.14: Provide an $\mathbf{O(n^3 \cdot \log(n))}$ algorithm to compute $\mathbf{5}$SUM. According to current knowledge, this algorithm is optimal up to polylogarithmic factors.

Homeworks 10.15-10.17

Homework 10.15: Assume that a language $\mathbf{L}$ as well as its complement are both context-free. Note that such languages have a grammar in Greibach Normal form where every rule is of the form $\mathbf{A} \to \mathbf{aW}$ where $\mathbf{A}$ is a nonterminal, $\mathbf{a}$ a terminal and $\mathbf{W}$ a possibly empty word of nonterminals (one either does not consider the empty word or needs some exception handling). Prove that $\mathbf{L}$ has a nondeterministic linear time decider which decides the membership of words in $\mathbf{L}$ for all nonempty words.

Homework 10.16: Construct a nondeterministic decider for 5SUM which runs in time $\mathbf{O(n^{2.56})}$.

Homework 10.17: Modify Theorem 10.11 and its proof such that it shows that, assuming ETH, there is no $\mathbf{c}$ such that for every $\mathbf{k}$, $\mathbf{k}$SUM can be solved in time $\mathbf{O(n^c)}$.

Homeworks 10.18-10.20

Homework 10.18: Determine a time function $\mathbf{F}$ such that all problems $\mathbf{k}$SUM are solvable in time $\mathbf{o(F(n))}$ while, assuming ETH, 3SAT and KNAPSACK cannot be solved in time $\mathbf{O(F(n))}$. Provide this bound $\mathbf{F(n)}$ explicitly. Note that here for each $\mathbf{k}$SUM, the minimum $\mathbf{n(k)}$ from which onwards $\mathbf{k}$SUM can be solved in time $\mathbf{F(n)}$ or less, might depend on $\mathbf{k}$.

Homework 10.19: Assume a problem has an algorithm which uses the recurrence $\mathbf{Time(3n) = 8Time(n) + O(n^2)}$. What is, according to Theorem 10.1, its complexity. Upround a noninteger exponent to five digits.

Homework 10.20: Assume a problem has an algorithm which uses the recurrence $\mathbf{Time(3n) = 5Time(n) + O(n^2)}$. What is, according to Theorem 10.1, its complexity. Upround a noninteger exponent to five digits.

Homework 10.21 and 10.22

Use Theorem 10.1 for the following two homeworks. Here one wants to compute a function value $\mathbf{f(x)}$ with $\mathbf{x} \in \{\mathbf{0}, \mathbf{1}\}^{\mathbf{n}}$.

Homework 10.21: Assume a problem has an algorithm which computes $\mathbf{f(x)}$ by either nondeterministically switching to a $\mathbf{O(n^3)}$ subroutine or by doing $\mathbf{17}$ subroutines of size $\mathbf{n/4}$ of itself with an additional processing of $\mathbf{O(n^2)}$. Can the nondeterministic switching be eliminated by hardwiring one choice? What is the complexity then?

Homework 10.22: Assume a problem has an algorithm which computes $\mathbf{f(x)}$ by either nondeterministically choosing between $\mathbf{25}$ self-calls with data of size $\mathbf{n/3}$ or $\mathbf{60}$ self-calls of size $\mathbf{n/4}$. In both cases, there is additional runtime of $\mathbf{O(n^{5/2})}$ in the local routine. Can the nondeterministic choice be eliminated by hardwiring one of the choices? If yes, which one is better and what is the overall complexity (uprounded to five digits)?

Lecture 11

This lecture deals with the Polynomial hierarchy (PH) and its relation to PSPACE. It will furthermore introduce and investigate the role of oracles in computing. This oracles will in particular be considered for the notions of P, NP, CoNP, PH and PSPACE. Relationsships between relatived worlds will be studied.

Furthermore, some concrete problems other than quantified Boolean formulas will be exhibited for the lower levels of PH. They are formulated as integer expression problems where an integer expression is either given as a finite set or the sum of two expressions or the union of the two sets given by expressions. In particular inclusion and equality to an interval of integer expressions are key problems placed into PH.

Computations Relative Oracles

Definition 11.1: A Turing machine can use an oracle A as follows: It has an oracle tape which is an additional tape governed by the same size constraints as applies to the work memory of the machine and the machine has an oracle-evaluation command with the following constraints:

- Command is of form “If word in Oracle Then Goto X Else Goto Y” where X and Y are line numbers;
- The command checks whether the word on the oracle tape is in A;
- If it is in A then the machine erases the content of the oracle tape, scrolls to its beginning and goes to Line X;
- If it is not in A then the machine erases the content of the oracle tape, scrolls to its beginning and goes to Line Y.

All other commands are executed as usual and above conditional jumps are the only interaction with A.

The Polynomial Hierarchy

Definition 11.2: One defines inductively the complexity classes $\mathbf{\Sigma_k^P}, \mathbf{\Pi_k^P}$ as follows: $\mathbf{\Sigma_1^P}$ is NP and $\mathbf{\Pi_1^P}$ is CoNP.

$\mathbf{\Sigma_{k+1}^P}$ is $\mathbf{NP[\Sigma_k^P]}$, that is, NP with an oracle which is a $\mathbf{\Sigma_k^P}$ complete set. Furthermore, $\mathbf{\Pi_k^P}$ is the class of all languages whose complement is in $\mathbf{\Sigma_k^P}$.

Example 11.3: The class $\mathbf{\Sigma_2^P}$ is equal to NP[NP], more precisely, to the class NP[SAT] where SAT is the satisfiability problem and might be replaced by any other NP-complete problem.

Alternating Computations

An alternating computation can be viewed as a game between two players Anke and Boris. One player wants to get a word accpeted, the other wants to get it rejected.

A language L is in NP if there is a nondeterministic Turing machine recognising it such that the player Boris has a possibility to make the nondeterministic choices such that the word gets accepted.

A one-time alternation is if during the game, there might arise the situation that from now on the choices are made by the other player, say Anke. So first it are Boris moves, but during the game it can happen that the rules say from now on Anke moves. If Boris can move such that he wins, independent of what Anke does later, the word is in L; otherwise it is in the complement of L.

One-Time Alternation I

Theorem 11.4: A language $\mathbf{L}$ can be computed by a one-time alternating computation if it is in NP[NP].

Proof one-time alternation $\Rightarrow$ NP[NP]: If one has one-time alternation then one can consider subprograms given by a configuration which is an alternation-point and the question whether the second player can continue such that the input will be rejected.

Thus $\mathbf{x} \in \mathbf{L}$ iff there is a nondeterministic polynomial time computation leading to such a switching point where the oracle says that the configuration does not lead to a rejecting decision, whatever the other player does. Thus one can nondeterministically guess the first computation and then check with an NP-complete oracle whether after the switch the other player can force a rejection.

One-Time Alternation II

Proof NP[NP] $\Rightarrow$ one-time alternation: Assume that a nondetermistic machine with NP-oracle recognises L. Then one can guess a computation and for all positive answers to an NP-complete oracle, one also guesses the witness. Thus one can guess a nondeterministic computation ending in a condition that all negatively answered queries are correct. These can be checked by an alternation which then requires that the other player continues to play and if the opposing player cannot disprove that any of the negative NP-answers is incorrect, then the overall computation is correct. Thus one can transform every computation in NP[NP] into a one-time alternating computation, where first Boris makes the decision and, after the alternation point Anke checks whether she can disprove any of the negative answers of the NP-oracle. If so, she rejects and provides the witness.

Generalising the Result

Theorem 11.5: A language $\mathbf{L}$ is in $\mathbf{\Sigma^P_{k+1}}$ iff there is a $\mathbf{k}$-time alternating computation starting with player Boris such that $\mathbf{x} \in \mathbf{L}$ iff Boris can enforce that the computation accepts.

A language $\mathbf{L}$ is in $\mathbf{\Pi^P_{k+1}}$ iff there is a $\mathbf{k}$-time alternating computation starting with player Anke such that $\mathbf{x} \in \mathbf{L}$ iff Anke can enforce that the computation rejects.

Recall that $\mathbf{k}$-time alternating computations are nondeterministic computations controlled alternately by players Anke and Boris such that computations end up in an explicit ACCEPT or REJECT decision. There are at most $\mathbf{k}$ points in the computation where the control alternates from one player to the other. A word is in $\mathbf{L}$ iff player Boris can enforce that the computation accepts; a word is outside $\mathbf{L}$ iff player Anke can enforce that the computation rejects.

Alternation and Quantifiers

One can evaluate languages in $\mathbf{NP}[\mathbf{NP}]$ (one-time alternating computation) using quantifiers: if $\mathbf{x} \in \mathbf{L}$ then Boris can choose the decisions until the alternation point such that for all subsequent choices by Anke the computation still ends up in an accepting state. If $\mathbf{x} \notin \mathbf{L}$ then independently of what Boris did first, Anke can when it becomes her turn enforce that the computation ends up in an rejecting state.

In other words: $\mathbf{x} \in \mathbf{L}$ iff there are decisions of Boris such that the computation either accepts or goes to an alternating point and if the latter happens, for all subsequent decisions of Anke, the outcome is still accept.

In a formula: $\exists$ decisions by Boris $\forall$ decisions of Anke [the outcome is accept]. The decisions here are limited in number by a polynomial for each player and can be written as a polynomially sized $\exists\forall$ formula.

Integer Expressions

Definition 11.6: Additive expression for integers are either finite sets of natural numbers given as an explicit list of binary integers or sums of expressions defined with $\mathbf{L+H=\{x+y : x \in L, y \in H\}}$ or unions of expressions. The length of an expression is the number of symbols needed to write it down.

Example 11.7: The integer expression $\mathbf{(\{0,1\}+\{0,2\}) \cup (\{4,8\}+\{0,8\})}$ generates the set $\mathbf{\{0,1,2,3,4,8,12,16\}}$.

Every finite set has an integer expression listing its elements in order; however, there are often more compact way of writings expressions.

Example 11.8: The set $\mathbf{\{0,1,2,3,4,5,6,7\}}$ can be written as $\mathbf{\{0,1\}+\{0,2\}+\{0,4\}}$ using six instead of eight numerical constants. In general, for intervals of length $\mathbf{n}$ one needs only an expression with $\mathbf{O(\log(n))}$ numbers.

Complete Problems in PH I

Each level of PH has complete problems; however, if one problem is complete for the whole PH, PH has only finitely many distinct levels.

Definition 11.9: The problem
$\exists \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n} \, \forall \mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_n} \, \exists \mathbf{z_1}, \mathbf{z_2}, \ldots, \mathbf{z_n} \, [\phi(\mathbf{x_1}, \ldots, \mathbf{z_n})]$
where ϕ is a Boolean formula and all variables are Boolean is $\mathbf{\Sigma_3^P}$ complete and this method relativises to all levels of PH.

Proposition 11.10: It is an NP-complete problem to decide whether a given binary number is a member of a given additive integer expression.

For this, one translates KNAPSACK instances into additive expressions. That is, a subset of $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$ can add to $\mathbf{y}$ iff $\mathbf{y} \in \{\mathbf{0}, \mathbf{x_1}\} + \{\mathbf{0}, \mathbf{x_2}\} + \ldots + \{\mathbf{0}, \mathbf{x_n}\}$.

Complete Problems in PH II

Theorem 11.11 [Wagner 1987]: The problem whether an additive integer expression L is a not necessarily proper subset of a further expression H is Π_2^0-complete.

The condition is in Π_2^0 as one can for each number formable by selecting members from possible choices of number-sets in subexpressions of L find a nondetermistic choice how to form the same number in H. So one has to show completeness.

For completeness one starts with a Boolean formula where variables are first universally and then existentially quantified. The intermediate values in the formula can also all be replaced by existentially quantified variables which takes as values these intermediate values. The variables in the formula are furthermore connected by assignments to the or of two variables or to the not of one variable.

Complete Problems in PH III

Now consider a conditions of the type

$$\mathbf{z = x \vee y}$$

and one represents them by the following equations – including a further auxiliary variables $\mathbf{u}$ to make it possible that the sum gets the value $\mathbf{2}$:

$$\mathbf{x + y + u + 2 \cdot \neg z = 2.}$$

If $\mathbf{x + y \geq 1}$, then $\mathbf{z}$ must be $\mathbf{1}$ and $\mathbf{u = 2 - x - y}$; if $\mathbf{x + y = 0}$ then $\mathbf{z, u}$ must both be $\mathbf{0}$. For the $\mathbf{k}$-th such equation, one reserves the $\mathbf{k}$-th decimal digit from below in the coding.

Complete Problems in PH IV

The following example gives how the numbers are coded in:

$$\mathbf{z = x \vee y, z' = \neg x \vee y, z'' = \neg x \vee \neg z.}$$

The corresponding numerical equations are these:

$$\mathbf{x{+}y{+}u{+}2\neg z = 2, \neg x{+}y{+}u'{+}2\neg z' = 2, \neg x{+}\neg z{+}u''{+}2\neg z'' = 2.}$$

Now one makes for each variable, say $\mathbf{x}$, two numbers, one representing the literals $\mathbf{x}, \neg\mathbf{x}$ as follows:
$\mathbf{x_{pos} = 110, x_{neg} = 001}$. For the case of $\mathbf{z}$, one must take into account the scaling factor $\mathbf{2}$ when the variable stands on the other side of the original equation:
$\mathbf{z_{pos} = 000, z_{neg} = 201}$. As $\mathbf{z}$ does not occur in the middle equation, both $\mathbf{z_{pos}, z_{neg}}$ have the middle digit $\mathbf{0}$.

Complete Problems in PH V

The last step is encoding that some variables are chosen by a second expression. The way to do this is to introduce additional digits in the number and to code the value into them as $\mathbf{1}$ or $\mathbf{2}$. So when $\mathbf{x}, \mathbf{y}$ are universally quantified, one would $\mathbf{x_{pos}}, \mathbf{x_{neg}}$ by $\mathbf{10110}, \mathbf{20001}$ on the right side expression and similarly for $\mathbf{y}$. The left side expression is then this: $\{\mathbf{00222}\} + \{\mathbf{10000}, \mathbf{20000}\} + \{\mathbf{01000}, \mathbf{02000}\}$. So the first two decimal digits code whether $\mathbf{x}, \mathbf{y}$ have to be positive or negative and the other three ensure that the value of the sum is the intended target $\mathbf{2}$. Thus $\mathbf{L} \subseteq \mathbf{H}$ holds iff for every possible choice of $\mathbf{x}, \mathbf{y}$ on the left side the right side can be made to sum up to to $\mathbf{222}$ on the three lower digits when the same choice of $\mathbf{x}, \mathbf{y}$ is taken.

This technique allows to code initial universal quantification into integer expression comparison.

Complete Problems in PH VI

Theorem 11.12 [Wagner 1987]: The following problem is Σ_3^P-complete: Given an integer expression H and a number y, is there an x with $\{x+1, x+2, \ldots, x+y\} \subseteq H$.

Note that here y is given as a binary or decimal number, not as a unary number.

Remark 11.13: In Proposition 11.10 and Theorems 11.11 the expressions can be formed from finite sets with adding expressions only, where in Theorem 11.9, one quantifies over the x existentially and thus goes up from Π_2^P to Σ_3^P. If $y = h + 2^3$ with $0 < h \leq 2^3$, then one can write the expression L on the left side as

$$L = \{x+1\} + \{0,1\} + \{0,2\} + \{0,4\} + \{0,h\}.$$

This holds for general 2^ℓ in place of 2^3.

PH and Randomness I

Definition 11.14: A language $\mathbf{L}$ is in BPP iff there exists a polynomial time algorithm $\mathbf{F}$ which uses random-bit sequences $\mathbf{b} = \mathbf{b_0 b_1 \ldots b_{p(n)}}$ such that, for all $\mathbf{x}$ and for $\mathbf{2/3}$ of these random-bit sequences $\mathbf{b}$, $\mathbf{F(x, b) = L(x)}$. That is, $\mathbf{F}$ outputs with probability $\mathbf{2/3}$ the right bit $\mathbf{L(x)}$. Here $\mathbf{L(x) = 1}$ for $\mathbf{x \in L}$ and $\mathbf{L(x) = 0}$ for $\mathbf{x \notin L}$.

Remark 11.15: One can increase the threshold to any constant. The idea is to run the algorithm $\mathbf{c}$ times with random bits; then one takes the majority of the outcomes. For example, the majority of three outcomes is correct with probability $\mathbf{(8 + 3 \cdot 4)/27 = 20/27}$ and false with probability $\mathbf{(1 + 3 \cdot 2)/27 = 7/27}$.

Proposition 11.16: BPP is closed under Boolean operations (union, intersection, complement).

PH and Randomness II

Theorem 11.17 [Lautemann 1983; Gacs (Sipser) 1983]: BPP is a subclass of Σ_2^P, that is, of NP[NP].

Algorithm: Meyer and Stockmeyer [1975] showed that $L \in \Sigma_2^P$ iff there is a set H in P and there are polynomials r, s such that for all words x,
$x \in L \Leftrightarrow \exists y \in \{0,1\}^{r(|x|)} \forall z \in \{0,1\}^{s(|x|)} \, [(x,y,z) \in K]$.

By Remark 11.15, one can assume that the probability of a false decision is strictly below 2^{-n}; let $s(n)$ be the uniform bound on the number of random bits used for the overall process. Let $k(n)$ be the uprounded value of $2s(n)/n$ and $r(n) = k(n) \cdot s(n)$. Let $y(n)$ be a vector of $k(n)$ strings $y_1, \ldots, y_{k(n)}$ of length $s(n)$ and z be just vector of $s(n)$ bits. Now $(x,y,z) \in K$ iff $L(x, y_h + z)$ accepts for some $h \in \{1, \ldots, k(n)\}$ where $y_h + z$ is the bitwise exclusive or and $L(x,u)$ is the BPP-algorithm on x with random-bits u.

PH and Randomness III

Lautemann [1983] used now calculations to show the following:

If $\mathbf{x} \in \mathbf{L}$ then there are $\mathbf{y_1, \ldots, y_n}$ such that for each $\mathbf{z}$ there is a $\mathbf{h}$ such that $\mathbf{L(x, y_h + z)}$ accepts. If one selects $\mathbf{y_1, \ldots, y_{k(n)}}$ at random, then the probabity that a $\mathbf{z}$ makes all of $\mathbf{L(y_h + z)}$ to reject is at most $\mathbf{2^{-n \cdot k(n)}}$ which is below $\mathbf{2^{-s(n)}}$, so one might expect that no string rejects all combinations. Thus a random choice of the $\mathbf{y_1, \ldots, y_{k(n)}}$ would make $\mathbf{(x, y, z) \in K}$ for all $\mathbf{z}$. Lautemann did some careful calculations in order to replace the probability arguement by a counting arguement.

PH and Randomness IV

If $\mathbf{x} \notin \mathbf{L}$ then one has to show that for all $\mathbf{y_1, \ldots, y_n}$ there is a $\mathbf{z}$ such that for all $\mathbf{h}$ $\mathbf{L(x, y_h + z)}$ rejects.

Here now the probability that $\mathbf{L(x, y_h + z)}$ accepts is below $\mathbf{2^{-n}}$ by assumption on the machine. The union probability such that all $\mathbf{L(x, y_h + z)}$ accepts for at least one $\mathbf{h}$ is below $\mathbf{k(n) \cdot 2^{-n}}$. As $\mathbf{k(n)}$ is a polynomial, this probability is below $\mathbf{1/2}$ for all sufficiently long inputs $\mathbf{x}$ and for no sufficiently long input $\mathbf{x} \notin \mathbf{L}$ and all $\mathbf{y_1, \ldots, y_{k(n)}}$ there is a $\mathbf{z}$ with $\mathbf{L(x, y_h + z)}$ rejecting for all $\mathbf{h}$.

For the small $\mathbf{x}$ one can patch $\mathbf{K}$ to hardwire that $\mathbf{K(x, y, z) = L(x)}$. Then the formula

$$\mathbf{x \in L \Leftrightarrow \exists y \in \{0, 1\}^{r(n)}\, \forall z \in \{0, 1\}^{s(n)}\, [K(x, y, z)]}$$

holds for all $\mathbf{x}$ and $\mathbf{L \in \Sigma_2^P}$.

As BPP is closed under complement, $\mathbf{L \in \Pi_2^P}$, too.

When P[A] = PSPACE[A] I

Theorem 11.18 [Baker, Gill and Solovay]: There is an oracle $\mathbf{A}$ with $\mathbf{P[A] = NP[A] = PSPACE[A]}$.

Proof: Let $\varphi_0, \varphi_1, \ldots$ be an enumeration of all PSPACE computable functions with $\mathbf{p_e}$ be the polynomial bound on the space used and $\mathbf{k_e}$ be the number of tape symbols of the work alphabet of the Turing machine $\varphi_\mathbf{e}$. The set $\mathbf{A} = \{\mathbf{1^d 0^{k_d \cdot p_d(|x|)} 1xb} : \varphi_\mathbf{d}(\mathbf{x}) = \mathbf{b}\}$ is a PSPACE-complete set. First one shows that $\mathbf{P[A]}$ contains all of PSPACE.

This is done by computing, for fixed $\mathbf{e}$, for each input $\mathbf{x}$ the string $\mathbf{1^e 0^{k_e \cdot p_e(|x|)} xb}$ with $\mathbf{b = 0, 1}$ and then asking which of these strings is in $\mathbf{A}$. That $\mathbf{b}$ where the answer is $\mathbf{1}$ is then output.

This algorithm is in polynomial time relative to $\mathbf{A}$, however, the larger the space bound of $\varphi_\mathbf{e}$ is, the longer the algorithm takes to write the oracle query.

When P[A] = PSPACE[A] II

Furthermore, one can see that $\mathbf{A}$ is in LINSPACE: On input $\mathbf{1^e 0^{k_e \cdot p_e(|x|)} 1xb}$, the machine can use the space of size of the input to simulate $\varphi_{\mathbf{e}}$, without loss of generality, this Turing machine does not have more than $\mathbf{e}$ states. Thus one can use the space of size $\mathbf{e}$ to store the Turing machine states and other things; the space of size $\mathbf{k_e \cdot p_e(|x|)}$ can be used to simulate the content on the Turing machine tape of the machine $\varphi_{\mathbf{e}}$. So $\mathbf{A}$ is in LINSPACE. It follows that $\mathbf{PSPACE[A] \subseteq PSPACE}$ and therefore $\mathbf{P[A] = PSPACE[A]}$.

As $\mathbf{P[A] \subseteq NP[A] \subseteq PSPACE[A]}$, all three are equal.

When P[B] and NP[B] Differ

The following gives the summary of further oracle results.

Theorem 11.19 [Stockmeyer 1976]: If $\mathbf{\Sigma_k^P[A] = \Pi_k^P[A]}$ then $\mathbf{PH[A] = \Sigma_k^P[A]}$.

Theorem 11.20 [Baker, Gill and Solovay 1975]: There is an oracle $\mathbf{B}$ such that $\mathbf{NP[B] \neq CoNP[B]}$ and thus $\mathbf{NP[B] \neq P[B]}$.

Theorem 11.21 [Heller 1984]: There is an oracle $\mathbf{C}$ with $\mathbf{\Sigma_2^P[C] \neq \Pi_2^P[C]}$, thus $\mathbf{P[C], NP[C], NP[NP[C]]}$ are all different.

Theorem 11.22 [Yao 1985; Hastad 1986]: There is an oracle $\mathbf{D}$ relative to which all levels of the polynomial hierarchy are different. For each such oracle $\mathbf{D}$, $\mathbf{PH[D] \subset PSPACE[D]}$.

PH and PP Relativised

Repetition: Recall that PP just says that there is a probabilistic Turing machine which decides $\mathbf{L}$ in the following way: For each $\mathbf{x} \in \mathbf{L}$ the computation accepts with at least probability greater than $\mathbf{1/2}$ and for each $\mathbf{x} \notin \mathbf{L}$ it accepts with probability strictly smaller than $\mathbf{1/2}$. Here the computations use $\mathbf{p(n)}$ random bits and steps on an input of size $\mathbf{n}$ and one out of $\mathbf{2^{p(n)}}$ computations abstains from a decision to avoid a tie (the remaining number is odd).

Theorem 11.23 [Toda 1991]: Let $\mathbf{A} \in \mathbf{PH}$ and $\mathbf{B} \in \mathbf{PP[A]}$. Then there is a $\mathbf{C} \in \mathbf{PP}$ such that $\mathbf{B} \in \mathbf{P[C]}$.

In particular the class $\mathbf{PP}$ is not a subclass of the polynomial hierarchy unless the latter collapses to a finite level. Furthermore, $\mathbf{B} \in \mathbf{P[C]}$ is also known as "$\mathbf{B}$ is polynomial time Turing reducible to $\mathbf{C}$."

Homeworks 11.24-11.29

There are two ways of measuring the size of an integer expression: (a) by the number of symbols used to write it down; (b) by the number of numerical parameters (binary numbers) occurring in the expression when writing it down. This homework asks for finding integer expressions as small as possible with respect to (b) for the following sets $\mathbf{U}, \mathbf{V}, \mathbf{W}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}$:

Homework 11.24: $\mathbf{U} = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{8}, \mathbf{9}, \mathbf{10}, \mathbf{27}, \mathbf{28}, \mathbf{29}, \mathbf{30}\}$.

Homework 11.25: $\mathbf{V} = \{\mathbf{8}, \mathbf{9}, \mathbf{16}, \mathbf{17}, \mathbf{32}, \mathbf{33}\}$.

Homework 11.26: $\mathbf{W} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}\}$.

Homework 11.27: $\mathbf{X} = \{\mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{8}, \mathbf{9}, \mathbf{10}, \mathbf{12}, \mathbf{16}\}$.

Homework 11.28: $\mathbf{Y} = \{\mathbf{0}, \mathbf{3}, \mathbf{6}, \mathbf{9}, \mathbf{12}, \mathbf{15}, \mathbf{18}, \mathbf{21}, \mathbf{31}, \mathbf{34}\}$.

Homework 11.29: $\mathbf{Z} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{27}, \mathbf{32}\}$.

Homework 11.30

Homework 11.30: Assume that $\mathbf{L}$ is a problem in BPP such that the BPP algorithm makes on inputs of length $\mathbf{n}$ an error with probability at most $\mathbf{2^{-2n-25}}$ with $\mathbf{s(n)}$ random bits.

A company wants to make a decider for this problem which uses as few random bits as possible. The idea is to use a data base and whenever a word $\mathbf{x}$ of length $\mathbf{n}$ comes up for the first time (no word of length $\mathbf{n}$ before) then the machine draws $\mathbf{s(n)}$ random bits and writes the random string as $\mathbf{u_n}$ into the data base. All subsequent words $\mathbf{x'}$ of length $\mathbf{n}$ will output $\mathbf{L(x', u_n)}$, that is, the value of the decider with the random string $\mathbf{u_n}$.

Calculate the overall probability that this algorithm ever misclassifies $\mathbf{L}$ on any input.

Homeworks 11.31 and 11.32

Homework 11.31: Assume that $\mathbf{L}$ has a decider $\mathbf{M}$ which uses $\mathbf{r(n)}$ random bits in order to decide every input with error probability bounded by $\mathbf{1/4}$. Provide a polynomial $\mathbf{k(n)}$ such that, when repeating the computation $\mathbf{k(n)}$ times on inputs of length $\mathbf{n}$, the error probability is at most $\mathbf{2^{-2n-25}}$; the method then uses $\mathbf{k(n) \cdot r(n)}$ random bits on length $\mathbf{n}$.

Homework 11.32: Assume that the algorithm from Homework 11.30 is fed with inputs in length-lexicographic order. Is there an input $\mathbf{x}$ such that when having processed $\varepsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}, \mathbf{000}, \ldots, \mathbf{x}$ the amount of random bits used so far is on average below $\mathbf{0.00001}$ bits per data item?

Lecture 12

This lecture will be about PSPACE and the classes beyond including undecidable problems.

PSPACE equals alternating polynomial time. Furthermore, by Savitch's theorem, a language $\mathbf{L}$ in NSPACE$(\mathbf{p(n)})$ for some polynomial $\mathbf{p}$ is also in SPACE$(\mathbf{p(n) \cdot p(n)})$ and thus in PSAPCE, so NSPACE = PSPACE.

PSPACE has a further characterisation, namely by interactive proof systems.

Many complexity classes can be characterised by their complete sets as the class of all problems LOGSPACE reducible to such a complete set.

PSPACE-Complete Problems I

The language taken as oracle A in Theorem 11.11 is a PSPACE-complete set. This completeness is obtained by coding. There are more natural problems.

Example 12.1: The set of all optimal moves for all $n \times n$ Reversi situations is a PSPACE-complete set.

$$\begin{array}{|c|c|c|c|}\hline . & . & . & . \\ \hline . & X & X & . \\ \hline . & O & O & . \\ \hline . & . & . & . \\ \hline \end{array}, \quad \begin{array}{|c|c|c|c|}\hline . & O & O & O \\ \hline . & O & X & . \\ \hline . & O & X & . \\ \hline . & . & . & X \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{|c|c|c|c|}\hline . & O & O & O \\ \hline . & O & X & . \\ \hline . & X & X & . \\ \hline X & . & . & X \\ \hline \end{array}$$

The starting position and sample move in 4×4 Reversi. The game starts in the centre. A player can move if he can play his new piece on an empty field (.) such that some pieces of the opponent are between the new and some old piece vertically, horizontally or diagonally which are turned.

PSPACE-Complete Problems II

Reversi players move alternately and pass only if they cannot move; the game ends when both players are unable to move. The aim of Reversi is to have at the end more (and if possible much more) pieces than the opponent.

Theorem 12.2: Some two-player board games have an optimal strategy which is in PSPACE. Some games like Reversi satisfy that their strategy is PSPACE complete.

Theorem 12.3 [Chandra, Kozen and Stockmeyer 1981]: Everything what can be computed in Alternating Polynomial Time (APOLYTIME or AP) is in PSPACE and vice versa.

Theorem 12.4: Let ϕ be a Boolean formula, for example a 3CNF formula. Now evaluating formulas of the type $\exists \mathbf{x_1} \forall \mathbf{x_2} \exists \mathbf{x_3} \forall \mathbf{x_4} \ldots \forall \mathbf{x_{2n}} \, [\phi(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{2n}})]$ is PSPACE complete. Note the main difference to 3SAT is that one uses alternating quantifiers instead of existential ones.

PSPACE-Complete Problems III

The formula from Theorem 12.4 can be evaluated in PSPACE by recursive calls. For this one always takes the first quantified variable in a formula and replaces it by $\mathbf{0}$ and $\mathbf{1}$, respectively:

Algorithm Decide(α):
If $\alpha = \exists \mathbf{x}\,[\beta(\mathbf{x})]$ then Return Decide$(\beta(\mathbf{0})) \vee$ Decide$(\beta(\mathbf{1}))$.
If $\alpha = \forall \mathbf{x}\,[\beta(\mathbf{x})]$ then Return Decide$(\beta(\mathbf{0})) \wedge$ Decide$(\beta(\mathbf{1}))$.
Otherwise β consists of clauses where all literals are replaced by either $\mathbf{0}$ or $\mathbf{1}$. If each clause contains a literal evaluating to $\mathbf{1}$ then return $\mathbf{1}$ else return $\mathbf{0}$.

The local memory of each call is linear (size of calling formula plus local variables) and the depth of the calls is also linear, thus the whole formula can be evaluated in polynomial space which is sufficient to store all the stacks and local memories involved with the recursive calls.

PSPACE-Complete Problems IV

Assume that $\mathbf{L} \in$ PSPACE via an algorithm using $\mathbf{p(n)}$ space which uses $\mathbf{2^{q(n)}}$ steps at most for polynomials $\mathbf{p, q}$. Note that $\mathbf{q(n) \in O(p(n))}$ as it only must take care of the number of different ways to write the tape on the $\mathbf{p(n)}$ positions and to choose the Turing state. Now to verify that a computation verifying that $\mathbf{L(x) = b}$ starting with a start configuration encoding $\mathbf{x, b}$ and taking at most $\mathbf{2^{q(n)}}$ steps until a unique configuration for halting and accepting one does the calls the procedure on the next slide with parameter $\mathbf{k = 0, 1, \ldots, q(n)}$.

Each round will consist of Anke putting configurations α, β and then Boris selecting the middle γ and then the round will either conclude with checking or the game will go into the next round. The overall number of rounds is $\mathbf{q(n) + 1}$.

PSPACE-Complete Problems V

Round $\mathbf{k}$: Given α, β by initial call or choice of Anke in last round.

If $\mathbf{q(n) - k < 1}$ then one verifies that that one can go in one step or less from α to β. If so or if player Anke did not stick to the rules during the game (that is, did in some round not take α, β from the two choices of the previous round where applicable) then one accepts else one rejects.

If $\mathbf{q(n) - k \geq 1}$ then first player Boris guesses a middle configuration γ.

Second player Anke selects

$$(\alpha', \beta') \in \{(\alpha, \gamma), (\gamma, \beta)\}$$

and passes it to round $\mathbf{k+1}$ for the new (α, β) there.

PSPACE-Complete Problems VI

If the PSPACE-computation goes to accept, then Player Boris will always select the configuration γ in the middle of the two configurations α, β of the computation. It is an invariant that the steps to computation to be covered per round is $\mathbf{2^{q(n)-k}}$. Thus, when $\mathbf{q(n) - k = 0}$ then at most one steps from α to β what is then checked easily.

If the computation starting with α does not lead to the accepting and halting configuration β within $\mathbf{2^{q(n)}}$ steps then each selection of Boris will have that either one cannot go from α to γ in $\mathbf{2^{q(n)-1}}$ steps or one cannot go from γ to β in $\mathbf{2^{q(n)-1}}$ steps. By prudent choices, Anke passes this nonreachability on from round to round until in the last round, one cannot go from α to β in at most one step which can be checked easily.

PSPACE-Complete Problems VII

The overall space usage of this is $\mathbf{4 \cdot p(n) \cdot (q(n) + 1)}$ symbols as each local round just stores the local α, β, γ plus some small extra space for $\mathbf{n}, \mathbf{k}$ and other ingredients. Thus the local memory can be translated into $\mathbf{s(n)}$ bits for some polynomial $\mathbf{s(n)}$ where Boris selects γ and Anke selects α', β' for the next rounds. One quantifies the bits selected by Anke universally and those selected by Boris existentially and furthermore evaluates the whole thing with a large formula to make sure that Anke really takes the border configurations from the previous two choices and that in the last step the computations from α to β can be done in at most one computation step.

Some quantified Boolean formula encodes these conditions and witnesses PSPACE-hardness of the formula in Theorem 12.4.

Interactive Proofs

Description 12.5: An interactive proof system is given by a pair (Prover, Verifier) where the prover has no restrictions in computation time and space and the verifier works in probabilistic polynomial time. Prover and verifier communicate by messages. They work on a language $\mathbf{L}$ as follows: For input $\mathbf{x}$ the prover proposes a value $\mathbf{b}$ for $\mathbf{L}(\mathbf{x})$ and supplies in messages the verifier with proof ideas and answers to the verifiers queries. If the verifier accepts correct suggestions and proofs and rejects incorrect suggestions and proofs both with probability at least $\mathbf{2/3}$ then $\mathbf{L}$ is in IP.

Theorem 12.6 [Adi Shamir 1992]: A language $\mathbf{L}$ is in PSPACE iff it is in IP.

Theorem 12.7 [Jain, Ji, Upadhyay and Watrous 2009]: Even if one uses a Quantum Computer for the verifier, the complexity class equal to IP is still PSPACE.

12.8 The Elementary Hierarchy

Let $\mathbf{F_0(n) = Poly(n)}$ be the collection of all polynomially bounded functions in one input $\mathbf{n}$ and let $\mathbf{F_{k+1}(n) = 2^{F_k(n)}}$.

Definition 12.8: The elementary hierarchy is given as $\mathbf{P \subseteq PSPACE \subseteq TIME(2^{Poly(n)}) \subseteq SPACE(2^{Poly(n)}) \subseteq TIME(2^{2^{Poly(n)}}) \subseteq SPACE(2^{2^{Poly(n)}}) \subseteq \ldots}$ where the levels $\mathbf{2k, 2k+1}$ can be defined using $\mathbf{F_k}$ as $\mathbf{TIME(F_k(n))}$ and $\mathbf{SPACE(F_k(n))}$.

Whether the inclusion between neighbouring levels is proper, one does not know, but the level $\mathbf{k+2}$ is always a proper superclass of the level $\mathbf{k}$ and no set inside level $\mathbf{k}$ can be complete for level $\mathbf{k+2}$ or higher. The level 2 is called EXPTIME and the level 3 is called EXPSPACE, the level 4 is called DOUBLEEXPTIME and the level 5 is called DOUBLEEXPSPACE. Level $\mathbf{k+1}$ can also be obtained by using alternating computations from the level $\mathbf{k}$.

Elementary Problems

Example 12.9: Presburger Arithmetic. Set of all true quantified formulas over integers using $+, -, <, =, >$, Boolean connectives, integer constants and variables. Situated between DOUBLEEXPTIME and DOUBLEEXPSPACE. A formula in Presburger Arithmetic is $\forall \mathbf{x}\, \exists \mathbf{y}\, [\mathbf{x} = \mathbf{y} + \mathbf{y} \vee \mathbf{x} = \mathbf{y} + \mathbf{y} + \mathbf{1}]$ ("every integer is even or odd").

Example 12.10: Checking whether a position in $\mathbf{n} \times \mathbf{n}$ checkers is winning, losing or draw is EXPTIME-complete.

Example 12.11: Computing the complement of a regular expression is DOUBLEEXPTIME-hard. This is due to the size-increase in the formula constructed, just writing it down takes this time in the worst case.

Example 12.12: Given a deterministic Turing machine $\mathbf{M}$, an input $\mathbf{x}$ and a binary number $\mathbf{y}$, to check whether $\mathbf{M}(\mathbf{x})$ halts within $\mathbf{y}$ steps is EXPTIME-complete.

The Exponential Hierarchies

The exponential hierarchy differs from the elementary hierarchy by using oracles from the Polynomial Hierarchy and allowing exponential time computations relative these oracles. Here one makes distinctions between allowing E-time or EXP-time where E-time means the runtime is of the form $2^{O(n)}$ while EXP-time means that the runtime is of the form $2^{Poly(n)}$. E-time classes are not closed under LOGSPACE reductions, as those can increase the size polynomially; thus one often considers EXP-time classes.

The E-hierarchy is ETIME(P), ETIME(NP), ETIME(NP[NP]), $\ldots$ and the k-th level allows ETIME computation relative to an oracle of the k-th level of the polynomial hierarcy. Similarly one defines the EXPTIME-hierarchy as EXPTIME(P), EXPTIME(NP), EXPTIME(NP[NP]), $\ldots$ and the properties of both are different; both hierarchies collapse when PH collapses.

The LOOP Programming Language

Definition 12.13: A LOOP program uses registers as an addition machine and has the following primitive operations: Set to 0, Increment, Copy value from one variable to another; Loop x Do $\ldots$ End. The data type of LOOP programs are natural numbers including zero.

Note that when the Loop command reads out the x before going into the loop and does the number of iterations given by the entrance value of x; subsequent changes to x are ignored.

With some tricks one can make operations like "$y = 1$; Loop x Do Begin $y = 0$ End." This assigns to y the value 1 iff x has the value 0. This can be generalised to subtraction which is not predefined and must be defined using loops.

Example

While the LOOP programs for adding and multiplying are standard, those for subtraction are more tricky.

Computing $\mathbf{x_0 = \max\{0, x_1 - 1\}}$.
$\mathbf{x_3 = 0}$; $\mathbf{x_0 = x_3}$;
Loop $\mathbf{x_1}$ Do Begin
  $\mathbf{x_0 = x_3}$;
  $\mathbf{x_3 = x_3 + 1}$ End.

Computing $\mathbf{x_0 = \max\{0, x_1 - x_2\}}$.
$\mathbf{x_4 = x_1}$; $\mathbf{x_0 = x_4}$;
Loop $\mathbf{x_2}$ Do Begin
  $\mathbf{x_3 = 0}$;
  Loop $\mathbf{x_4}$ Do Begin
    $\mathbf{x_4 = x_3}$;
    $\mathbf{x_3 = x_3 + 1}$ End;
  $\mathbf{x_0 = x_4}$ End.

The Grzegorczyk Hierarchy

Description 12.14: The Grzegorczyk Hierarchy is the hierarchy of functions of the natural numbers to natural numbers which can be computed by LOOP programs and those which can be written by LOOP programs with $\mathbf{k}$-fold nesting of loops form the hierarchy level $\mathcal{E}_{\mathbf{k}}$. The Level $\mathbf{0}$ cannot be explained with LOOP programs without any Loop-command and the Level $\mathbf{1}$ are those functions which add some inputs; the level $\mathcal{E}_{\mathbf{2}}$ include also functions like multiplication. The level $\mathcal{E}_{\mathbf{3}}$ includes exponential functions and iterated exponential functions like $\mathbf{x} \mapsto \mathbf{2}^{\mathbf{2}^{\mathbf{x}}}$ and is called the level of the elementary functions.

A function from natural numbers to natural numbers is called primitive recursive iff it can be written with LOOP programs, that is, is on some level of the Grzegorczyk Hierarchy.

General Programming Paradigm

Functions which can be described by a computer program, for example that of an Addition Machine or Turing Machine, are called "partial-recursive functions"; the add-on "partial" means that it might happen that on some input $\mathbf{x}$, the computer program runs forever and therefore the function (the program's output) is undefined. All primitive recursive functions are also partial-recursive, but not vice versa, as all primitive recursive functions are total, that is, everywhere defined. There are also some total functions which are not primitive recursive, for example the Ackermann function:

$$\mathbf{Ack(x,y)} = \begin{cases} \mathbf{y+1} & \text{if } \mathbf{x=0;} \\ \mathbf{Ack(x-1,1)} & \text{if } \mathbf{x>0 \wedge y=0;} \\ \mathbf{Ack(x-1,Ack(x,y-1))} & \text{if } \mathbf{x>0 \wedge y>0.} \end{cases}$$

A numbering of all primitive recursive functions with one input variable is another such example.

Halting Problem

Fact 12.15: There is a computer program $\mathbf{u}$ with two inputs $\mathbf{e}, \mathbf{x}$ such that for each one-input program $\mathbf{p}$ there is an index $\mathbf{e}$ so that $\mathbf{u}(\mathbf{e}, \mathbf{x}) = \mathbf{p}(\mathbf{x})$ for all $\mathbf{x}$ (with $\mathbf{u}(\mathbf{e}, \mathbf{x})$ being undefined when $\mathbf{p}(\mathbf{x})$ is undefined).

Theorem 12.16: There is no computer program $\mathbf{u}'$ such that for all $\mathbf{e}, \mathbf{x}$, $\mathbf{u}'(\mathbf{e}, \mathbf{x})$ is defined and for all total $\mathbf{p}$ there is an $\mathbf{e}$ with $\mathbf{u}'(\mathbf{e}, \mathbf{x}) = \mathbf{p}(\mathbf{x})$.

Assume that $\mathbf{u}'$ exists and then consider $\mathbf{p}$ computing $\mathbf{p}(\mathbf{x}) = \mathbf{u}'(\mathbf{x}, \mathbf{x}) + \mathbf{1}$.

Corollary 12.17: A two-input computer program is either not total or it cannot capture (as rows) all total one-input computer programs. In particular, the question whether $\mathbf{u}$ above halts (= is defined) on inputs $\mathbf{e}, \mathbf{x}$ cannot be decided by a total computer program.

Halting Problem and Polynomials

Definition 12.18: (a) Recursive Enumerable: If an algorithm can enumerate (in whatever speed and order) the members of a set, the set is called recursive enumerable. The halting problem is an example.
(b) Diophantine Sets: A set $\mathbf{L}$ of integers is Diophantine iff there is a polynomial $\mathbf{p}$ in $\mathbf{k+1}$ variables for some $\mathbf{k}$ such that $\mathbf{x \in L \Leftrightarrow \exists y_1, \ldots, y_k \, [p(x, y_1, \ldots, y_k) = 0]}$.

Example 12.19: The set $\mathbf{\{x : \exists y, z > 2 \, [y^2 - x \cdot z^2 = 1]\}}$ is Diophantine. A Diophantine is recursive enumerable since one can go through all $\mathbf{k+1}$-tuples of $\mathbf{(x, y_1, \ldots, y_k)}$ and whenver the polynomial evaluates to $\mathbf{0}$ on this tuple, one enumerates $\mathbf{x}$.

Theorem 12.20 [Yuri Matiyasevich 1972]: Every recursively enumerable set of integers is Diophantine.

The Arithmetical Hierarchy

One can investigates quantifier hierarchies on polynomial expressions. Here bounded quantifiers with finite range are ignored. So one has the following for sets of numbers as well as for sets of tuples of numbers:

- A set is Σ_0 if it can be defined with bounded quantifiers only and if it is defined by an expression using $+$, $-$, $\cdot$, $=$, $<$, $>$ and access to variables and number constants. Such sets coincide with the Π_0 sets.
- A set A is Σ_{k+1} iff there is a Π_k set B with $h + h'$ variables such that $(x_1, \ldots, x_h) \in A$ if and only if $\exists y_1 \ldots \exists y_{h'} \, [(x_1, \ldots, x_h, y_1, \ldots, y_{h'}) \in B]$.
- A set A is Π_{k+1} iff there is a Σ_k set B with $h + h'$ variables such that $(x_1, \ldots, x_h) \in A$ if and only if $\forall y_1 \ldots \forall y_{h'} \, [(x_1, \ldots, x_h, y_1, \ldots, y_{h'}) \in B]$.

Diophantine sets (= recursively enumerable sets) are Σ_1.

Usage of Oracles I

Theorem 12.21: A set $\mathbf{A}$ is in $\mathbf{\Sigma_{k+1}}$ if and only if it is recursively enumerable relative to a $\mathbf{\Sigma_k}$ set $\mathbf{B}$. A set $\mathbf{A}$ is in $\mathbf{\Pi_{k+1}}$ if and only if its complement is recursively enumerable relative to a $\mathbf{\Sigma_k}$ set $\mathbf{B}$.

Example 12.22: The following sets are in $\mathbf{\Sigma_0}$:
(a) $\{(\mathbf{x},\mathbf{y}) : \mathbf{x^2 - 2y^2 = 1}\}$ – Equation set given by a two-variable polynomial.
(b) $\{(\mathbf{x},\mathbf{y}) : \mathbf{0 \leq x \wedge x^2 < y \wedge y < x^2 + 2x + 2}\}$ – Pairs where $\mathbf{x}$ is the largest nonnegative integer strictly below the square-root of $\mathbf{y}$.
(c) $\{\mathbf{x} : \exists \mathbf{v} \leq \mathbf{x}\, \exists \mathbf{w} \leq \mathbf{x}\, [\mathbf{x = v^2 + w^2}]\}$ – Numbers which are the sum of two integer squares.

Example 12.23: The following sets are $\mathbf{\Sigma_1}$:
(a): Halting problem; (b): Range of some recursive function; (c): All $\mathbf{\Sigma_0}$ sets including $\emptyset$; (d): All Diophantine Sets.

Usage of Oracles II

Example 12.24: The following sets are Σ_2:
(a): All sets enumerated relative the halting problem as an oracle;
(b): Set of all computer programs which compute a function with a finite range;
(c): Set of all computer programs which work only on finitely many inputs;
(d): Set of all computer programs which do not halt on some input.

Example 12.25: The following sets are Π_1:
(a): Set of all computer programs which do not halt on input 0;
(b): Complement of a recursively enumerable set;
(c): Set of all computer programs terminating after $x^2 + 1$ steps latest on each input x.

Blum Complexity Measures

Definition 12.26 [Manuel Blum 1967]: A pair (φ, Φ) is a Blum complexity measure where $\varphi_0, \varphi_1, \ldots$ is a numbering of all partial-recursive functions and $\Phi_e(x)$ is defined iff $\varphi_e(x)$ outputs some value and the set of triples $\{(e, x, t) : \Phi_e(x) = t\}$ is decidable, that is, satisfies that some computer program can check whether $\Phi_e(x) = t$.

A complexity class C is given by a family $f_0, f_1, \ldots$ of total recursive functions such that for every $g \in C$ there there are indices d, e such that $g = \varphi_d$ and, for all x, $\Phi_d(x) \leq f_e(x)$.

Example 12:27: The complexity class of all polynomial time computable functions is given by the family of all polynomials and $\Phi_e(x)$ being the computation time which φ_e uses to compute $\varphi_e(x)$. If $\Phi_e(x)$ is the computation space, then one obtains the class of all PSPACE-computable functions. The additional requirement that the functions g are $\{0, 1\}$-valued leads to the classes P (for time) and PSPACE (for space).

Properties of Complexity Measures

Theorem 12.28 [Gap Theorem by Trakhtenbrot 1967 and Borodin 1972]: Given a recursive function $\mathbf{g}$ with $\mathbf{g(n) \geq n}$ for all $\mathbf{n}$, there is a Time bound $\mathbf{t}$ such that, for all $\mathbf{n}$, $\mathbf{DTIME(t(n)) = DTIME(g(t(n)))}$. So the extra time computed by $\mathbf{g}$ does not help. This goes also with abstract Blum complexity.

Theorem 12.29 [Blum Speedup Theorem 1967]: Given a two-place recursive function $\mathbf{f}$ there exists a total recursive $\mathbf{g}$ such that for each program $\mathbf{i}$ for $\mathbf{g}$ there is a further program $\mathbf{j}$ for $\mathbf{g}$ such that, for all but finitely many $\mathbf{n}$, $\mathbf{f(n, \Phi_j(n)) \leq \Phi_i(n)}$.

Proposition 12.30: Every complexity class has a numbering of the functions / sets in the class; this numbering is outside the class, as the bit-wise exclusive or of $\mathbf{1}$ and the diagonal function does not occur in the numbering.

Homeworks 12.31-12.34

Homework 12.31: Provide two levels in the elementary hierarchy such that NEXPTIME sits between these levels. The levels should be as close to each other as possible.

Homework 12.32: If the levels DOUBLEEXPSPACE and DOUBLEEXPTIME of the elementary hierarchy are the same, does this imply anything on other levels to be the same?

Homework 12.33: A language $\mathbf{L}$ is called tally iff for every two $\mathbf{x}, \mathbf{y}$ of the same length it holds that $\mathbf{L}(\mathbf{x}) = \mathbf{L}(\mathbf{y})$. Prove the following: A tally language is in EXPSPACE iff the set $\{$ binary $\mathbf{n} : \mathbf{0}^{\mathbf{n}} \in \mathbf{L}\}$ is in DOUBLEEXPSPACE.

Homework 12.34: Is there a language which is inside the Elementary Hierarchy and hard for all levels with respect to LOGSPACE-many-one reductions? Prove the answer.

Homeworks 12.35-12.38

A simplified version of the Loop Programming Languages uses all commands of addition machines except for GOTO which is replaced by For-loops and If-Then-Else statements; the borders and the loop variable of a For-loop cannot be modified inside the loop, the loop variable is always incremented after each run of the loop and the loop is not entered when the lower bound is above the upper bound. Write programs using this programming language for the following functions – they are thus primitive recursive. The input $\mathbf{x}$ is a positive integer.

Homework 12.35: $\mathbf{f(x) = x^{x^x}}$.

Homework 12.36: $\mathbf{g(x) = x! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot x}$.

Homework 12.37: $\mathbf{h(x) = g(g(g(g(g(x)))))}$.

Homework 12.38: $\mathbf{k(x)}$ is the first prime number above $\mathbf{2^x}$.

Homeworks 12.39-12.42

Homework 12.39: It there a total recursive two-input function $e, x \mapsto \varphi_e(x)$ such that for each total recursive two-input function $e, x \mapsto \psi_e(x)$ there is a d such that φ_d differs from each ψ_e on some input x?

Homework 12.40: Provide a partial-recursive function f such that the set $\{(x, y) : f(x) = y\}$ is decidable (= recursive) but the set $\{x : f(x)$ is defined$\}$ is undecidable.

Homework 12.41: Provide a numbering of all primitive recursive functions.

Homework 12.42: Explain why there is a number of all primitive recursive functions (which are all total) but not one of all total recursive functions.

On Final Exam

The Final Examination is on Tuesday 6 May 2025 from 17:00 to 19:00 hrs.

The syllabus comprises all twelve lectures of this term and includes the material tested in the midterm examination; of the twelfth lecture in particular the first parts are relevant. There will be 10 questions each scoring 6 points testing both general knowledge on computational complexity and problem solving skills. It is a closed book exam with one page helpsheet.