Contents

Course description

The aim of the course is to enable the student to

- Apply formalisms of formal languages in order to formulate decision problems precisely
- Construct finite automata, regular expressions, push-down automata and context-free grammars as elements in an algorithmic solution of more complicated problems.
- Decide the complexity of new problems based on knowledge of the complexity of important examples of problems from the course.
- Judge whether a given problem may be solved by a computer or is undecidable.
- Argue that problems are NP-complete.
- Judge the possibility to develop an approximation algorithm for a given NP-hard optimization problem.
- Give lower bounds for the complexity of problem that are similar in nature to those studied in the course.

These competencies are important both when one wishes to develop new algorithms for a given problem and when one wants to judge whether a given problem may be possible to solve efficiently (possibly only approximately) by a computer.

The course builds on the knowledge acquired in the courses DM507 Algorithms and data structures and DM551 Algorithms and probability.

The course forms the basis for doing a bachelor project as well as elective candidate level courses containing one or more of the following elements: complexity of algorithms, approximation algorithms and computability.

Together with courses as above this course also provides a basis for doing a masters thesis on algorithmic and complexity theoretic subjects.

In relation to the competence profile of the degree it is the explicit focus of the course to:

- Give the competence to analyze complexity of (decision) problems.
- Give knowledge about the computational power of different models of computation.
- Enable the student to construct finite automata and regular expressions for simple languages.
- Enable the student to construct push-down automata and context-free grammars for simple languages.
- Equip the students with important tools to prove that a given language cannot be recognized by a finite automation, a push-down automaton or a Turing machine.
- Enable the student to prove lower bounds for the complexity of algorithms for a given problem.
- Enable the student to develop new approximation algorithms.
- Give the student important tools for proving that a given decision problem is NP-complete or undecidable.

Aims

- Judge the complexity of (decision) problems.
- Judge the computational power of various models of computation.
- Construct finite automata and regular expressions for simple languages.

- Construct push-down automata and context-free grammars for simple languages.
- Prove that a given language, which in nature resembles those from the course, cannot be recognized by a finite automaton, a push-down automaton or a Turing machine.
- Prove lower bounds for the complexity of algorithms for a given problem which in nature resembles those from the course.
- Design new approximation algorithms for a given problem which in nature resembles those from the course.
- Prove that a given decision problem which in nature resembles those from the course is NP-complete or undecidable.

Chapter 1

Questions 2018

1. Finite automata and regular languages

Introduction

I'm going to talk about Finite automata and regular languages.

Finite automata

Finite automata is the simplest computational model that works via states and transitions, and therefor uses extremely limited memory. Using this model we can recognize and formulate regular languages, This can be a simple tasks a finding a substring, or used as a tool for designing more complex systems. A Finite atomata is defined as a tuple containing:

- The set of states Q,
- The known alphabet Σ,
- The transition function $\delta : Q \times \Sigma \longrightarrow Q$
- the start state q_1
- and the set of accept states F.

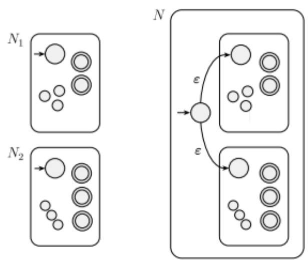
Regular languages

A regular languages is a sequence of letters in some alphabet defined by $\sum$ the empty alphabet is defined by the empty set $\varnothing$, and contains letters and the empty string ϵ. they are also closed under the union $\cup$, concatination $\cap$, and kleene star *, and the proceeding order is $^*, \cap, \cup$, A language is Regular is a Finite automata recognizes it.
Add definitions

Deterministic And Non-deterministic

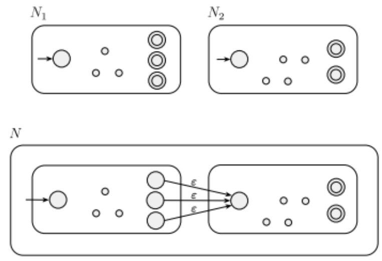
The difference between deterministic and non-deterministic is the way they operate. they recognize the same class of languages as a NFA can be convert into a DFA and the same goes the other way around, it is not a efficiency operation as the algorithm is exponential. there is however some benefits to both. where the non-deterministic preforms branching and could potentially benefit the execution wrt. size, runtime. or recourse consumption.
The correct term here would that that a DFA Can simulate a NFA

Regular languages Closure under



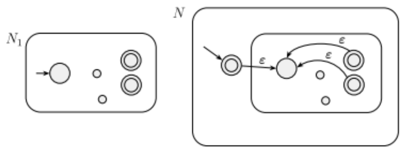
Union
Proof by construction of a NFA that recognize $A \cup A$,
We make a new start stat q_0 and make a ϵ transition to M_1 and M_2
The new machine is $q_0 \cup Q_1 \cup Q_2$
The accept states are $F = F_1 \cup F_2$
The new transition function is

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \epsilon \\ \varnothing & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$



Concatenation
We built a automata M that is the concatenation of M_1 and M_2 we simply start with a new start state M_1 and for each accept state in M_1 we go to the start state of M_2 $A_1 \circ A_2$
$Q = Q_1 \cup Q_2$
out star state is the same start state as q_1 in M_1
the accept states F_2 are the accept states for M_2
our transtion funciton is

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q2\} & q \in f_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$



Klein star
we need a new state q_0 that is also added to F, and we make a transition from all states in F to q_0
$Q = q_0 \cup Q$
the state q_0 is the new start state.
$F = F \cup q_0$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in f_1 \text{ and } a = \epsilon \\ \{q_1\} & q = q_0 \text{ and } a = \epsilon \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

Pumping lemma for regular languages

The pumping lemma is a way for us to proof if a language is regular. The theorem for the pumping lemma states that the 3 conditions for the lemma is

- for each $i \geq 0, xy^iz \in A$
- $|y| > 0$
- $|xy| \leq p$ where p is the pumpking length.

$0^n1^n|n \geq 0$
The way you do this is to assume it's regular and make a counter argument. with the pumping lemma we have 3 conditions that our counter argument most meet. the first case we can pump the language to have more 1's than 0s or(2) the other way around.. the third(3) is that we can get out of order letters if we pump a string containing both 0s and 1s,

Example -> Pumping lemma or convert NFA to DFA

TODO exam with pumping lemma

2. Pushdown automata and context-free languages

Introduction

I'm going to talk about Pushdown automata and context free languages. These topics are used in compilers to parse a programming language and is a useful tool in language processing as well as when interpreting one language to an other.

Pushdown automata

A pushdown automata is a type of automaton that employs a stack to help with the computation, this allows the PDA to recognize and run Context free grammars as well as the languages within(regular languages).
The formal definition is:

- The set of states Q,
- The known alphabet Σ (Sigma)
- The stack alphabet Γ (Gamma)
- The transition function $\delta(delta) : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \longrightarrow Q$
- the start state q_0
- the set of reject states states F

Context free grammar

A CFG consist of a collection of substitution rules, a CFG operate on a set of variables, terminals and a designated start symbol.
The formal definition is

- The finite set of variables V
- The finite set of terminals Σ disjoint from V
- The finite set of rules R
- The start symbol $S \in V$

Exsample

- $A \to 0A1$
- $A \to B$
- $B \to \#$

This above grammar should generate the string $0^n\#1^n$ $n \in \mathbb{N}$.
A different examples that show ambiguity is the set of variables and terminals {num,A,-}
With the rules

- $A \to A - A$
- $A \to num$

for the above grammar if we apply it to the string
1-2-3 we can get the output 2 or -4 depending on how the grammar is parsed. (1-2)-3 vs 1-(2-3)
We can handle this ambiguity in two manners either we introduce the terminals (and) to the language or we can rewrite or rules as to not have this issue.

Chomsky normal Form

Chomsky normal form is a simplified form that we can convert our grammars into which enables to decide things like if a string is generated by a grammar in polynomial time 2n-1
The steps to convert a gramma to CNF is to:

1. eliminate all ϵ productions.
2. Eliminate all productions where RHS is one variable
3. Eliminate all productions longer than 2 variables
4. move all terminals to productions where RHS is one terminal.

Example with CFG -> CNF

We start in the initial state
$A \to BAB|B|\epsilon$
$B \to 00|\epsilon$

From here we put a new start state S

$S \to A$
$A \to BAB|B|\epsilon$
$B \to 00|\epsilon$

From here we can eliminate the first ϵ

$S \to A$
$A \to BAB|B|\epsilon|BA|AB$
$B \to 00 \cancel{|\epsilon}$
From here we can eliminate the ϵ in our 2nd rule,

$S \to \cancel{|A}|BAB|\cancel{B}|BA|AB|\epsilon|CC$
$A \to BAB|\cancel{B}\cancel{|\epsilon}|BA|AB|CC$
$B \to \cancel{00}|\cancel{\epsilon}|CC$
$C- > 0$

Pumping lemma of non-CFL

The pumping lemma is a way for us to proof if a language is regular. The theorem for the pumping lemma states that the 3 conditions for the lemma is

- for each $i \geq 0, uv^ixy^iz \in A$
- $|vy| > 0$
- $|vxy| \leq p$

Example

The way you do this is to assume it's regular and make a counter argument. with the pumping lemma we have 3 conditions that our counter argument most meet.
$a^nb^nc^n|n \geq 0$

- if vy only contain the same symbol the string connot contain an equal number of letters as required
- when either v or y contains more than 1 type of symbol we get an out of order contradiction

3. Turing machines

Introduction

In this talk i'll talk about turning machines and their use in computer science, A turning machine is a theoretical form of computer in the same sense as the other models but it's the closets one to modern computers and we can use it for decide things wet. to run time, and compatibility of problems and algorithms.

What is a turing machine

What is a turing machine, The model itself is of a tape and a read/write head that can move left or right on the tape.

- Q is the set of states
- $\sum$ is the input alphabet not containing the blank symbol
- τ is the tape alphabet where $\in \tau$ and $\sum \subseteq \tau$
- $\delta : Q \times \tau \to Q \times \tau \times \{L, R\}$ is the transition function.
- $q_0 \in Q$ is the starte states
- $q_{accept} \in Q$ is the accept state
- $q_{reject} \in Q$is the reject state where $q_{reject} \neq q_{accept}$

Different types of Turing machines

All turning machines are equal in power, show why.

Multitape

The idea is to show that a multitape turing machine TM_m can be simulated on a equivalent single tape turing machine. this is done by storing the tapes on a single tape and discriminating them via the letter #,
And we use dot to mark the position of the read head on the underlying tapes. If we reach the condition where one of the simulated readheads reaches a # symbol on the right side of the simulated tape we write a ␣ and shift the contents of the tape by 1,

Nondetermanistic turning machine

A ND turing machine can be simulated by running a tree search to find accepting state for the desired configuration. and we'll want to approach this in a breath first search as using a dept first can result in following a infinite branch that never reaches a accept state, therefor it's better running a breath first as we're guaranteed to find a accept state should one exist but the issue of looping still exists. We further call a non deterministic turing machine a decider if all branches halt on all input.

Enumerators

Is a turing machine attached to a printer, basically it generates all possible outputs for a set configuration.

Theorem 3.21

A language is only turing recognizable if and only if some enumerator enumerates it.

The universal turing machine

Takes a description and some input w and simulates it, it may accept, reject or halt.
You could consider a universal turning machine the closets to a regular computer, except of the sense of the memory. A modern computer may have close to a tarabyte of memory but this still isn't the infinite memory of the turing machines.
A universal turning machine is a recognizer but not a decider as i recognizes A_{TM}
A_{TM} Is decidabl

Halting problem

we have a machine N and input w, we present a machine H that decides if N halts or not.

1. If N halts on input w, accepts
2. if N loops on input w, reject

We then build the machine H+, This new machine has the output modified, the idea is then to feed H+ with itself as as input and as the string.

1. If H+ with input <H+,H+> halts, we loop.
2. If H+ with input <H+,H+> doesn't halts accept.

Proof by contradiction

Assume that A_tm is decidable,
Let H be the tm that decides A_tm.

$$H(< M, w >) = \begin{cases} Accept, if m accepts w \\ Reject if m does not accept w or loops. \end{cases}$$

Using H, we can construct a new machine D
input to D is a turing machine. and problem that D runs is to check weather a machine would accept if given a input of itself.
output, do the opposite of what the input does.
Now we try to run D with itself as input.

$$D(< D >) = \begin{cases} Accept, if D does not accepts w \\ Reject if D accepts \end{cases}$$

We then we a paradox.

4. Decidability

The problem of Decidability is weather or not we can computer something, and what the limits of computers are. we need to use this to know if a problem is solvable or not with the current computational models we have.

Example

0^n1^n $n \geq 0$ and $n \in \mathbb{N}$ is not decidable for a DFA/DNA, but it is decidable for a PDA
But the grammar 0^n1^n $n \geq 0$ and $n \in \mathbb{N}$ is not decidable for a PDA, but is decidable by a TM

A_{DFA} Is decidable

The idea for this is to present a tm that simulates A_{DFA}
We build a TM M that takes input <B,w> where B is the DFA and w in our input.
1. M first determines if proprerly reprecents a DFA B and the string w, if not it rejects.
2. M then simulates B directly keeping track of how much input we have processed and what state we are in,
3. When M is done simulating eg, when we reach the end of w we accept if B accepts and rejects if we're in a non-accepting states

A_{NFA} is decidable

This can be proven by using the above example as a subroutine, and building make a TM N that first converts A_{NFA} to a DFA and then run the above problem on it.
NFA to DFA theorem chapter 1 theorem 39

A_{REX} is decidable

a regex can be converted into a NFA (theorem 1.54)
We present a turing machine that P that converts the regex into a NFA, we can then run N as a subroutine and show that RegEX is decidable.

A_{CFG} is decidable

we build a TM S, we take input <G,w> where G is a CFG and w is a string

1. convert G into Chomsky normal form.
2. list all derivations with 2n-1 steps, where n in the length of the string, except the case where n in length zero when then list all derivations with 1 step.
3. if any of these derivations generate w accept else reject.

This same tm can be used to show that all Context free languages are decidable

undecidability

Some problems are also undecidable, for example Uncountability, and not all turning machines are decidable.

countability and Uncountability

some finite sets we can simply count them, but how can we prove that one infinite set in larger than an other.
We use the following definition of countable, if our set has a finite size, or if there is a correspondence with $\mathbb{N}$

The set of odd numbers

The correspondence with off numbers is that we can simply map f(n) = 2n-1, where n is our regular numbers. here our odd numbers are also a subset of our natural numbers

The set of rational numbers

rational numbers can be expressed as $\{\frac{m}{n}|m$ and $n \in \mathbb{N}$
This set of countable infinite. this can by shown by using the The diagonalization method
The way that we can generate a list of all rational numbers. is to go over a matrix

M \ N	1	2	3	4	...
1	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	...
2	$\frac{2}{1}$	$\cancel{\frac{2}{2}}$	$\frac{2}{3}$	$\cancel{\frac{2}{4}}$	...
3	$\frac{3}{1}$	$\frac{3}{2}$	$\cancel{\frac{3}{3}}$	$\cancel{\frac{3}{4}}$	...
4	$\frac{4}{1}$	$\cancel{\frac{4}{2}}$	$\cancel{\frac{4}{3}}$	$\cancel{\frac{4}{4}}$	...
...	...	...	...	...	...

Irrational numbers

The set of irrational numbers is a uncountable infinite set. numbers like $\pi, \sqrt{2}, e$ and so on have a infinite numbers of digits.
between two ration numbers we have a infinite irrational numbers.
Proof by contradiction. assume that the set $\mathbb{I}$ is countable infinite.
this can be shown via:

The diagonalization method.

we can build a new number, that can't be in the table, which gives us a contradiction in the table.

some langues are not turing recognizable

We can show that we have a countable infinite amount of turing machines.

- every turing machine can be coded into a string, that have a finite length
- everything is either a valid turing machine or "garbage"
- Generate one string after the other
- check to see if it is a valid turing machine.

Proof (By diagonalization method)

We can show that there are uncountable infinite many strings.
We can show that we have a uncountable infinite amount of infinte strings over {0,1}
same idea as $\mathbb{I}$ uncountability, by flipping the bit. this we we can generate a new string that is infinite length and that is not in the table.
therefor: The number of languages is countably infinite.
and by the first example we showed that the set of all turing machines is countably infinite, and from this we have the corollary that the set of all turing-recognizable languages is countably infinite.
and we showed that the set of all langues is countably infinite.
We therefor have the corollary that some languages are not turing recognizable

A language that isn't turing recognizable

If a language L is decidable, then L is turing recognizable, and it's complement $\overline{L}$ is turing recognizable

- Every decidable language is turing recognizable
- want to recognize $\overline{L}$ just run the decider for L and give the opposite answer.

We run the machine for L and $\overline{L}$ in parallel. eg running one step on each until one of them reaches a accept state, and one of them has to be in one of them. and one or both of them has to halt.
If that machine for L accepts we accept, and if the machine for it's complement $\overline{L}$ accepts we reject. either way we'll always halt
with this it can be shown, that a language is only decidable if both it and it's complement are turing recognizable.
a language is co-turing recognizable is if it's complement is turning recognizable
we know that A_{TM} is turning recognizable,
and that A_{TM} is not decidable
Therefor $\overline{A_{TM}}$ is not turing recognizable
We can proof this as A_{TM} is not decidable.

5. Reducibility

Introduction

Reduciblity is the method of proving by mapping a more complex problem to a simpler one, and by this we can proof that if a problem is decidable or not. You must however be very careful when doing this at it can only be done if the problems are the same in nature.
The first thing we need to do when we are preforming this method is to have a problem that we can reduce to, This problem has to be mappable to our simpler problem and the property we want to map to has to be shown to be undecidable, in this case the A_{TM} problem will be used, the problem for A_TM is solved via the diagonalization method.

logic

With A_{TM} we can show that **P** is undecidable.
We assume that **P** is decidable.
We reduce the A_{TM} into a problem **P**
We then use the decidability of **P** to find an algorithm to decide A_{TM}
Built a TM to decide A_{TM} using the TM to decide P as a subroutine.
But we know that a decided for A_{TM} cannot exist, and therefor we reach a contradiction,

The halting problem

We can show that the halting problem is undecidable by using the undicidability of A_{TM}
We do this by presenting a TM R, that decides $Halt_{TM}$
R <M,w> where M is a turing machine and w is a string.

- If M accpts or rejects, accept,
- If M loops, reject.

We then present S<M,w> that decides A_{TM} where we use R as a subroutine
S <M,w> where M is a turing machine and w is a string.

- simulate if <M,w> halts by running it on R as a subroutine.
- If M loops, reject, else resume.
- simulate M on input w, and reject if it rejects and accept if it accepts.

Here we reach a contradiction, because we from the the diagonalization proof that A_{TM} is undecidable. this implies that S -> E -> halt isn't decidable either.

Mapping reducibility

Mapping reducibility means that a computable function exist that converts problem a to problem b, if we have such a function then that i called a reduction.
Theorem 5.22

$$\text{If } A \leq_m B \text{ and } B \text{ is decidable, then A is decidable}$$

Corollary 5.23

$$\text{If } A \leq_m B \text{ and } B \text{ is undecidable, then A is undecidable}$$

PCP to A_{TM}

The reasoning for this is that we can have some PCP that doesn't halt. and if it doesn't halt we can decide it.
Look at the at the following, we have A, B, C

- $A = \frac{10}{101}$

- $B = \frac{101}{011}$
- $C = \frac{011}{10}$

With the Above letter we can only start with A as it's the only valid start string, We can't start with B or C as the first above and below letter doesn't match.
When we start with A the only valid following symbol is B and a never ending row of Bs after this. so we can't enter a halting case. This is however more of a toy example, the bigger example involves showing the that MPCP is undecidable and from that mapping it to the PCP problem.

6. NP-completeness proofs – examples.

Introduction

NP completeness is a way for us to classify how hard problems are, we generally denote these problems in how long their run time is
What we care about in the run time and how we write it is using the big O notation. generally here we only scare about to the power of wrt. run time. as an example 2n and n^2 while the run time of 2n the 2 is quickly nullified as we just run the problems on more threads, or if we double the click speed. while the problems with a run time that is exponential we suddenly have bigger issues.

The class of NP problems.

NP is the class of languages that have deterministic polynomial time turing machine verifiers, but only have non-deterministic polynomial time turing machine solvers.
Definition 7.34
A problem is NP_complete if B is in NP
Every A in NP is polynomial time reducible to B
The above definition also follows that if we can solve one NP_complete problem in P time we can solve all NP_complete problems in P time. proving that

$$P = NP \tag{1.1}$$

The above is unproven.
Because of the before mentioned Definition we will assume that 3-sat is NP_complete as this can be decided via the cook levin theorem.

3-sat to qlique

qlique -> vertex cover

The same can be shown for all NP_complete problems, that solving one solves all of them, and if we solve one in p time we solve all of them in p time.

7. Proof that SATISFIABILITY is NP-complete (do not assume that there is a known NP-Complete problem — use the proof in Sipser's book).

NP completeness is a way for us to classify how hard problems are, we generally denote these problems in how long their run time is
What we care about in the run time and how we write it is using the big O notation. generally here we only scare about to the power of wrt. run time. as an example 2n and n^2 while the run time of 2n the 2 is quickly nullified as we just run the problems on more threads, or if we double the click speed. while the problems with a run time that is exponential we suddenly have bigger issues.

The class of NP problems.

NP is the class of languages that have deterministic polynomial time turing machine verifiers, but only have non-deterministic polynomial time turing machine solvers.
Definition 7.34
A problem is NP_complete if B is in NP
Every A in NP is polynomial time reducible to B
The above definition also follows that if we can solve one NP_complete problem in P time we can solve all NP_complete problems in P time. proving that

$$P = NP \tag{1.2}$$

The above is unproven.

Cook Levin Theorem

Cook-levin theorem - in sipset, presentation use slides on homepage.

8. Information-theoretic lower bounds

(lower bounds proven by counting leaves in decision trees), especially the average case bounds for sorting by comparisons.

Introduction

The study of lower bounds for algorithms is useful when studying the average case run time, in this case we do it by counting leaves in a decision tree,
The first thing we want to show is lower bounds for algorithms by counting leaves.
Insertion sort(2n), quicksort(2n) and merge sort(nlogn). (Worst case)

lower bounds

Using a decision trees we can show that the lower bound for a given comparison based sorting algorithm. the tree has n! nodes as a list of length n has n! permutations,

9. Adversary arguments – technique, examples.

10. Median problem – algorithm and lower bound.

11. Approximation algorithms

vertex cover

Chapter 2

Weekly notes

Most of these are from whiteboard, i decided to try and convert one to text, but the issue is that they don't contain any text explanation of whats going on. eg they're fine if you're precenting live with them but all the information that is mentioned doing the lecture isn't there so what you end up with is only half of what you need and without the other part this part the things showed in the notes are mainly undecidable.
This is something that should be brought to the attention of the lecturer, but at the current time i don't have time to look into it and may use this as exercise when i want to review this course at a later date.

May 4th

Maximum among n elements

- Compare first element x with second y and keep z = maxx,y
- Compare 2 with next and keep max etc.

find max in **n-1 comparisons**
Complexity is in # comparisons
Best possible: if x has not been compared with anything, then it can still be max.
Finding max and min: S set of n distinct numbers

Naive alg:

Find Max in S, call it z	n-1
Find Min in S-z.	n-2
Total run time of	2n-3

Different strategy:
Case 1:

Chapter 3

notes

2.4 Lower bounds for sorting by comparison of keys

In this section we derive lower bounds for the numbers of comparisons that must be done in the worst case and on the average(case) by any algorithm that sorts by comparison of keys. To derive the lower bounds we will assume that they keys in the list to be sorted are distinct.

2.4.1 Decision trees for sorting algorithms

Let n be fixed and suppose that the keys are $x_1, x_2, ..., x_n$. we will associate with each algorithm and positive integer n a (binary) decision tree that describes the sequence of comparisons carried out by the algorithm on any input of size n. let sort be an algorithm that sorts by comparisons of keys. each comparison has a two-way branch(since the keys are distinct), and we assume that sort has an output instruction that outputs the rearranged list of keys. The decision tree for sort is define inductively by associating a tree with each compare and output instruction as follows. The tree associated with an output instruction consists of one node labeled with the rearrangement of the keys. the tree associated with an instruction that compares keys x_i and x_j consists of a root labeled (i:j)m a left sub tree that is the tree associated with the instruction executed next if $x_i < x_j$. and a right sub-tree that is the tree associated with the instruction executed next if $x_i > x_j$. the decision tree for sort is the tree associated with the first compare instruction it executes. an example of a decision tree for n=3 is shown in figure 2.11

The action of sort on a particular input corresponds to following one path in its decision tree from the root to a leaf. the tree must have at least $n!$ leaves because there are $n!$ ways in which the keys may be permuted. since the unique path followed for each input depends only on the ordering of the keys and not on their particular values, exactly $n!$ leaves can be reached from the root by actually executing sort. we will assume that any paths in the tree that are never followed are removed. we also assume that comparison nodes with only one child are removed and replace by the child, and this pruning is repeated until all internal nodes have degree 2. the pruned tree represents an algorithm that is at least as efficient as the original one. so the lower bounds we derive using trees with exactly n! leaves and all internal nodes of degree 2 will be valid lower bounds for all algorithms that sort by comparison of keys. from no on we assume sort is described by such a tree.

The number of comparisons done by sort on a particular input is the number of inter nodes on the path followed for that input. this the number of comparisons done in the worst case is the number of internal nodes on the longest path, and that is the depth of the tree. the average number of comparisons done is the average of the length of all paths from the root to a leaf. (for exsample, for n = 3, the algorithm whose decition tree is shown in fig. 2.11 does tree comparisons in the worst case, and two and two thirds on the average(case).)

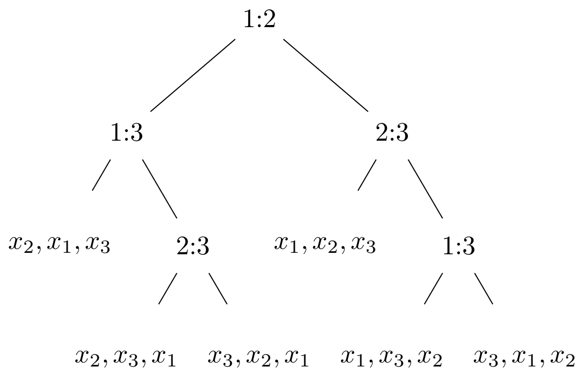


Figure 3.1: **Figure 2.11** Decision tree for a sorting algorithm, $n = 3$.

2.4.2 Lower bounds for worst case

To get a worst-case bound for sorting by comparisons, we derive a lower bound for the depth of a binary tree in terms of the number of leaves, since the only quantitative information we have about the decision tree is the number of leaves.

Lemma 2.4 Let l be the number of leaves in a binary tree and let d be it's depth,
The $l \leq 2^d$.

Proof. proof. a straightforward induction on d. □

Lemma 2.5 Let l and d be in lamma 2.4 then $d \geq \lceil logl \rceil$

Proof. proof. Taking logs of both sides of the inequality in lemma 2.4 gives $logl \leq d$. Since d is an integer, $d \geq \lceil logl \rceil$ □

Lemma 2.6 For a given n, the decision tree for any algorim that sorts by comparisons of keys has depth t least $\lceil logn! \rceil$.

Proof. Proof let $l = n!$ in lemma 2.5 □

So the number of comparisons needed to sort in the worst care is at least $\lceil logn! \rceil$. Our best sort so far is Merge sort, but how close is $\lceil logn! \rceil$ to $n\ logn$? there are several ways to estimate or get a lower bound for log n!. perhaps the simplest is to observe that:

$$n! \geq n(n-1)....\left(\lceil \frac{n}{2} \rceil\right) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \tag{3.1}$$

so

$$log\ n! \geq \frac{n}{2}\ lg\ \frac{n}{2} \tag{3.2}$$

which is in $\theta(nlogn)$, thus we see already that mergesort is of optimal order, to get a closer bound, we use the fact that

$$log\ n! = \sum_{j=1}^{n} log\ j \tag{3.3}$$

using eq. 1.9 we get

$$log\ n! \geq n\ log\ n - 1.5n \tag{3.4}$$

thus the depth of the decision tree is at least $\lceil n\ log\ n - 1.5n \rceil$

Theorem 2.7 any algorithm to sort n items by comparisons of keys must do at least $\lceil log\ n! \rceil$ or approximately $\lceil n\ log\ n - 1.5n \rceil$ key comparisons in the worst case. So Mergesort is very close to optimal,

there is some difference between the exact behavior of Mergesort and the lower bound. Consider the case where $n = 5$. Lower bound is $\lceil log\ 5! \rceil = \lceil log 120 \rceil = 7$ is the lower bound simply not good enough, or can we do better than Mergesort? the read is encouraged to try and find a way to sort five keys with only seven comparisons in the worst case.

2.4.3 Lower bound for average behavior

Chapter 4

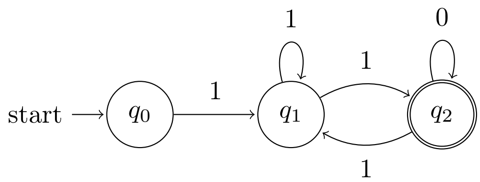
Exercises 2019

Week 1

page 84, question 1.7

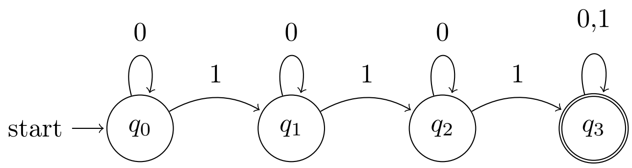
a

w|w begins with a 1 and ends with a 0



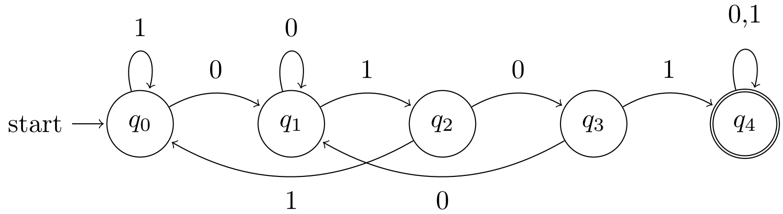
b

w|w contains at least 3 1's



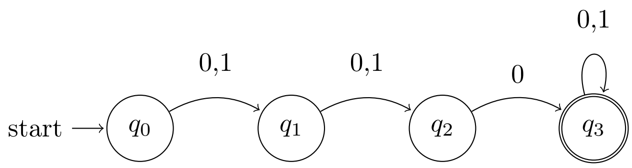
c

w|w contains the substring 0101, (i.e. w = x0101y for some x and y)



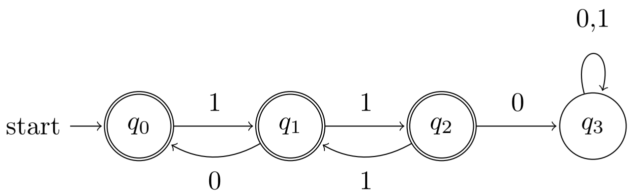
d

w|w bas at length of at least 3 and it's third symbol is 0



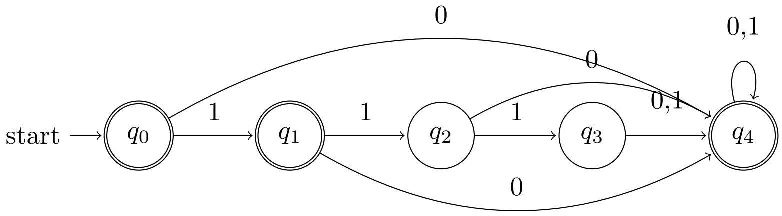
f

w|w doesn't contain the substring 001
Accepts any string that doesn't contain the substring 001, and loops in rejecting state if this state is found, was unsure if the looping was needed but included it if it was the case.



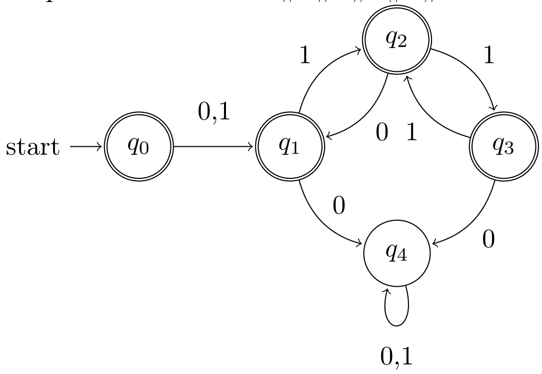
h

w|w is any string except 11 and 111
accepts any string that isn't 11, 111 including the empty string ϵ



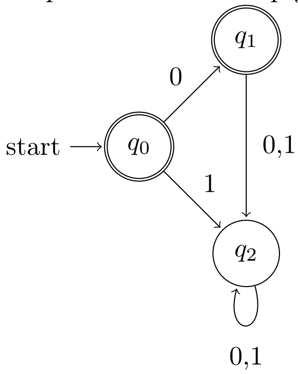
i

w|w every odd position of w is a 1
accepts all states where #1#1#1#1#1 is followed also accepts the empty string ϵ



k

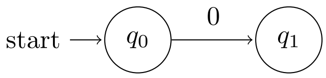
w|w is only the empty string and 0
accepts "0" and the empty string ϵ



page 84, question 1.7

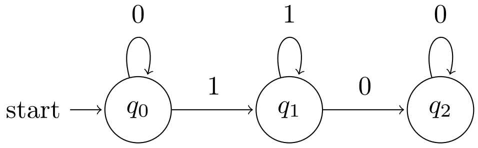
d

The language 0 with two states,



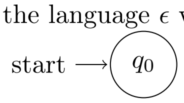
e

the language 0*1*0* with 3 states



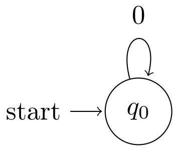
g

the language ϵ with 1 state



h

The language 0* with 1 state



Solve the following problem,

A man is travelling with a wolf (w) and a goat (g). He also brings along a nice big cabbage (c). He encounters a small river which he must cross to continue his travel. Fortunately, there is a small boat at the shore which he can use. However, the boat is so small that the man cannot bring more than himself and exactly one more item along (from w, g, c). The man knows that if left alone with the goat, the wolf will surely eat it and the goat if left alone with the cabbage will also surely eat that. The man's task is hence to device a transportation scheme in which, at any time, at most one item from w, g, c is in the boat and the result is that they all crossed the river and can continue unharmed.

a

Describe a solution to the problem which satisfies the rules of the "game". You may use your answer to (b) to find a solution.

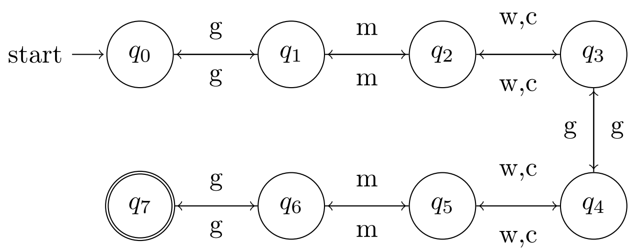
- First you carry the goat to the other side, and go back empty.
- The you ferry the wolf to the other side, and swap with the goat and bring the goat back.
- you then swap the goat with the cabbage and bring it to the other side.
- lastly you head back empty and bring the goat.
- you now have all the items on the other side of the river.

b

The string all of the valid moves are

$$\begin{aligned}(g(m(x(g(y(m(g)*)*)*)*)*)*)* \\ x, y = (w|c) \\ x \neq y\end{aligned} \tag{4.1}$$

This is due to the fact that it's legal moves in the game, like the man can bring the goat to the other side and bring it back too, it's bad move, but it's still a valid move.



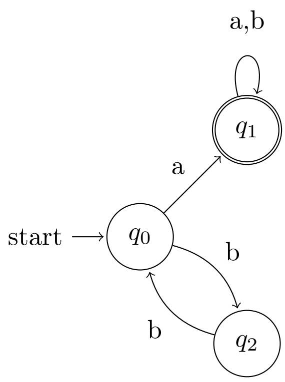
Week 2

page 86, question 1.16

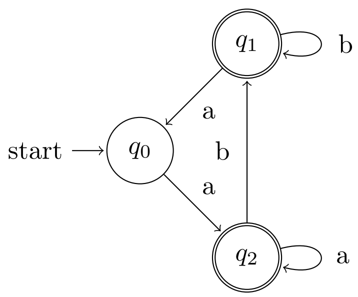
* is zero or more
! is one or more

a

it accepts (bb)*a!(a|b)*
the language 0*1*0* with 3 states



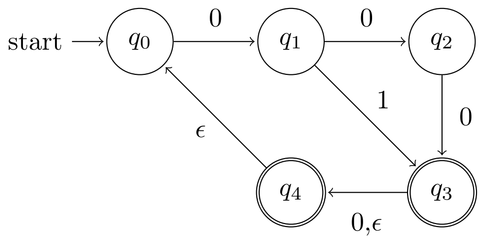
b



page 86, question 1.17

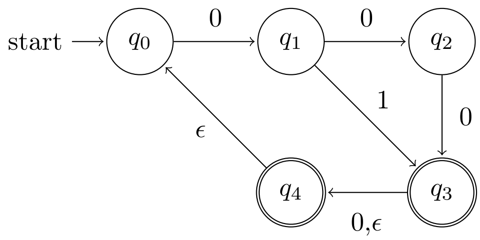
a

The first task is to make a NFA recognizing (01u001u010)*



b

After converting this to a DFA



page 86, question 1.18

Predefined terms

$$x = (0,1)^* \tag{4.2}$$

$$y = (0|1) \tag{4.3}$$

a

$1x0$

b

$x1x1x1x$

c

$x0101x$

d

$yy0x$

e

$(0(yy)*)|1y(yy)^*)$

f

$0^*(1+10)^*$

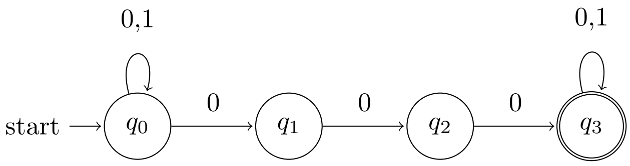
g

$yyyyy$

page 86, question 1.19 a

Convert the following regex to a NFA via lemma 1.55

$$(0 \cup 1)^* 000 (0 \cup 1)^* \tag{4.4}$$



page 86, question 1.20

For each of the following expressions give two strings that are and two that are not in the languages. assume that the alphabet is {a,b}

a

$$a^* b^* \tag{4.5}$$

member
aa, ab, aabb, aab, abbbb
not member
ba, abab,

b

$$a(ba)^* b \tag{4.6}$$

member
abab, ababababab,
not member
aaba baba

c

$$a^* \cup B^* \tag{4.7}$$

member
ab, aabb
not member
ba, aabba

d

$$(aaa)^* \tag{4.8}$$

member
aaa, aaaaaa
not member
a, aaaa

e

$$\overset{*}{\sum} a \overset{*}{\sum} b \overset{*}{\sum} a \overset{*}{\sum} \tag{4.9}$$

member
todo
not member
todo

f

$$aba \cup bab \tag{4.10}$$

member
todo
not member
todo

g

$$(\epsilon \cup a) b \tag{4.11}$$

member
ab, b
not member
abb, aab

h

$$(a \cup ba \cup bb) \overset{*}{\sum} \tag{4.12}$$

member
todo
not member
todo

page 86, question 1.21 b

The solution is

$$((a|b)(a, bb) * b(a)?)* \tag{4.13}$$

(a or b, followed by any number of (a,bb)* followed by a single b (as it matches uneven numbers of b and followed by 0 or 1 a*

page 86, question 1.29

a

For the task a we have the sting

$$0^n 1^n 2^n \tag{4.14}$$

the first contradiction is that when we have a string eg. 000111222 and we split it so we have the floowing, x = 000, y = 111, z = 222 and we pump y so we have the string xyyz this violates the first condition of the pumping lemma, as we'll have more 1s then 0s and 2s.
the string y only contains 1s which also causes a contradiction.
and the 3rd case if we have the string x = 00, y = 011 and z = 1222 where we'll get out of order letters so we'll again reach a contradiction.

b

For assignment b i find it a bit odd, i may misunderstand the exercise but w/e
We want to pump the language

$$a_2 = \{www|w \in \{a,b\}^*\} \tag{4.15}$$

But from my understanding is the * a kleen star? eg then one w and www are equivalent as {a,b} is all possibly strings in the library w. ? or is it understand such that w is equal to either a or b but any number of them eq $\{a|b\}^*$,
How i choose to intercept it for now is that w is equal to either the string a* or b* and if this is the case then some of the same argumentation as for task a is valid.
www can give us the string 111000111 and we can split it as follows. 111|1000|11111, This means that zyyx gives of a out of order 1 and we therefor get a contradiction wrt. to the pumping lemma.

page 88, question 1.30

The error There's a few things here, the first case from 1:73 fail right away as the string 0001111 is in the lang, the case of out of order fails as if we can get the string 00100111 ten we'll reach a contradiction but the case of

page 89, question 1.36

For the language w there exist a DFA that accepts if, for the revere language w^r the DFA which a reversed edges and the start state is now our accept state.

week 3

page 88, question 1.29(a),(b) and 1.30

Already done, see above.

page 89. question 1.35

As B is in bijection of A therefor it's a regular.

page 91. question 1.51

a

This can be proved via the pumping lemma. where you can get out of order 1 and/or 0s and that causes a contradiction.

b

Same as a wrt. our of order.

c

This is this is either the unbounded string 0 or 1.

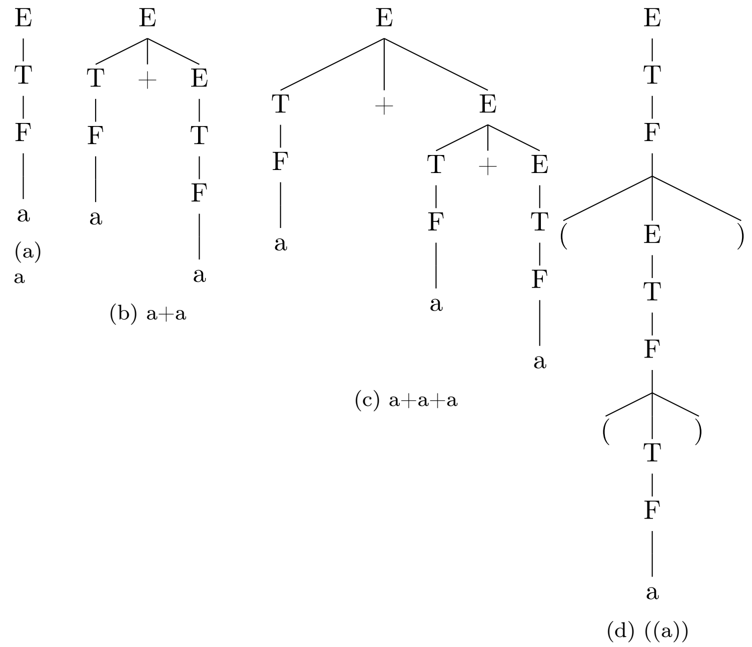
d

this is again wrt. out of order 0.1 eg. if we can get the string 01010 by umping as this is not in the lang.

page 154. question 2.2

Give parse trees and derivations for each string using the following Context free gramma (CFG)

$$E \rightarrow E + T|T$$
$$T \rightarrow E \times F|F$$
$$F \rightarrow (E)|a$$



(a) a

(b) a+a

(c) a+a+a

(d) ((a))

Figure 4.1: 2 Figures side by side

page 154. question 2.4

a

$S \rightarrow TTT$
$T \rightarrow E1E$
$E \rightarrow FTF$
$F \rightarrow 0|\epsilon$

b

$S \rightarrow 0F0|1F1$
$F \rightarrow F0F|F1F|\epsilon$

c

$S \rightarrow EFE$
$E \rightarrow FEF|EFF|FFE|\epsilon$
$F \rightarrow 0|1$

d

$S \rightarrow E0E$
$E \rightarrow FEF|\epsilon$
$F \rightarrow 0|1$

e

$S \rightarrow E$
$E \rightarrow FEF|GEG|\epsilon$
$F \rightarrow 0$ $G \rightarrow 1$

f

$S \rightarrow \epsilon$

2.6

Give the CFG that generates the following languages

b

The complment of the langauges $\{a^n b^n | n \geq 0\}$

$S \rightarrow FG$
$G \rightarrow GbG$
$F \rightarrow FaF$

d

For the third one it's a unbounded string with up to k alternations of 0*1*0*1*0*
This can simply be defined using the following grammar.
$S \rightarrow FG$
$E \rightarrow GE|FE|\epsilon$
$G \rightarrow 1G|\epsilon$
$F \rightarrow 0F|\epsilon$

2.14

We start in the initial state
$A \rightarrow BAB|B|\epsilon$
$B \rightarrow 00|\epsilon$

From here we put a new start state S

$S \rightarrow A$
$A \rightarrow BAB|B|\epsilon$
$B \rightarrow 00|\epsilon$

From here we can eliminate the first ϵ

$S \rightarrow A$
$A \rightarrow BAB|B|\epsilon|BA|AB$
$B \rightarrow 00 \cancel{|\epsilon}$
From here we can eliminate the ϵ in our 2nd rule,

$S \rightarrow \cancel{|A}|BAB|\cancel{B}|BA|AB|\epsilon|CC$
$A \rightarrow BAB|\cancel{B}|\cancel{\epsilon}|BA|AB|CC$
$B \rightarrow \cancel{00}|\cancel{\epsilon}|CC$
$C- > 0$

page 156, question 2.16

Show that the class of context free languages are closed under the regular operations Concatenation, union and Star
Using the following grammar

- $S_1 \rightarrow aS_1b$
- $S_1 \rightarrow \epsilon$
- $S_2 \rightarrow cS_2d$
- $S_2 \rightarrow \epsilon$

Concatenation

This is simply shown via
Making s start symbol S, and have the two languages follow each other S_1 S_2
$S \rightarrow S_1 S_2$

Union

This is simply shown via
Making s start symbol S,
$S \to S_1|S_2$

Star

This is simply shown via
Making s start symbol S,
$S \to S_1 S$

page 158, question 2.32

Let $A/B = \{w|wx \in A$ for some $x \in B\}$, Show that if A is context free and B is regular, then A/B is context free.
"Proof. We can augment the memory of the PDA recognizing the CFL with the DFA recognizing the regular langauge and run both machines in parallel. We accept iff both machines accept.**??**
note the intersection of two CFL's and not necessarily a CFL.!

page 158, question 2.38

As each step has a most 2 sub rules we can calculate that at step 1 we increasing the string length by 2n-1 and as all rules do this our end case is 2n-1

page 158, question 2.42

• 2.42 Hint for (d): first intersect the language with a suitably chosen regular language and then prove that the language you obtain is not context-free.

a

A break wrt. out of order letters, and not n counts of some letter

b

For the lang b is has to be in the size of wrt. $n + n^2 + n^3$ eg 3,14, 39, 84, 155 and you can pump it to a length not in this range,

c

i dont know

d

i dont know

week 4

2.58

let $\sum = \{0,1\}$ and let B be the collection of strings that contain at least one 1 in their second half, in other words,$B = \{uv|u \in \sum^*, v \in \sum^* 1 \sum^*$ and $|u| \geq |v|\}$

a

Give a PDA that recognizes B
$S \rightarrow XTX|X1$
$T \rightarrow XT|X$
$X \rightarrow 1|0$

b

Give a CFG that generates B Read the input from end to front, pushing every 0 you meet onto the stack. if a 1 is found you continue parsing but instead pop a letter from the stack until it's empty, if you reach the beginning of the input before the stack is empty you reject.

2002 program 2

Let $\sum$ be a finite alphabet and let $L \subset \sum^+$ be a regular language, Define a new language L'as the language of all words w $\in \sum^*$ such that w is obtained from a word in L by deleting the first letter, is the language L' regular as well? prove your answer