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.

Show a illustration of a state diagram

Regular languages

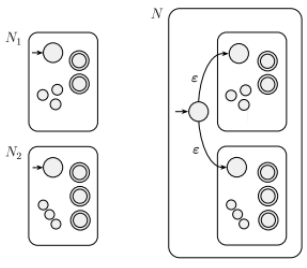
A regular languages is a sequence of letters in some alphabet defined by $\sum$ the empty alphabet is defined by the empty set $\varnothing$, and contains letters and the empty string ϵ. they are also closed under the union $\cup$, concatination $\cap$, and kleene star *, and the proceeding order is $^*, \cap, \cup$, 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.
The correct term here would that that a NFA can be converted into a DFA but this DFA may have n^2 many states.

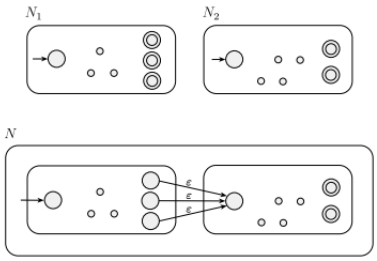
Regular languages Closure under

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



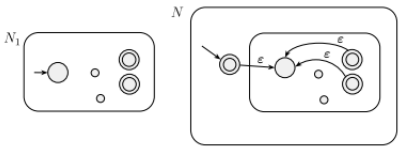
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \epsilon \\ \varnothing & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

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.
$F = F \cup q_0$



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

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

Language 0^n1^n where $n \geq 0$
For this example we can show 3 contradictions.

1. out of order 0 or 1 if our pumping string contains both 1 or 0
2. wrong number of 1s if our pumping string only contains 1s
3. wrong number of 0s if our pumping string only contains 0s

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

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

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

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 cannot 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,
The way a step from step one is simulated is that we start at cell 0 on both tapes. so does our master pointer.
We then scan from the beginning to the first tape marker, and read whats at the first read head. here after we scan forward until we see the next marker. this is repeated n times where n is the number of tapes. after this first scan, we scan to the start of the tape until we reach the beginning of the tape.
The cost for doing this in worst case is N*k where n is the size of our input and k is the number of tapes.

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 $0^n1^n0^n$ $n \geq 0$ 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 $\{\frac{m}{n} | m \text{ and } n \in \mathbb{N}$
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	$\frac{2}{1}$	~~$\frac{2}{2}$~~	$\frac{2}{3}$	~~$\frac{2}{4}$~~	...
3	$\frac{3}{1}$	$\frac{3}{2}$	~~$\frac{3}{3}$~~	~~$\frac{3}{4}$~~	...
4	$\frac{4}{1}$	~~$\frac{4}{2}$~~	~~$\frac{4}{3}$~~	~~$\frac{4}{4}$~~	...
...	...	...	...	...	...

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 countable infinite 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 $\overline{A_{TM}}$ 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.

SATISFIABILITY

What is it. string of boolean algebra that we want to find a solution to that make it return true.

Cook Levin Theorem

The cook levin proof states.

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.
Merge sort(nlogn). (Worst case)

run times

Theta

lower bounds for worst case

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. to sort a list we need to compare the n elements in a list. we can think of this as a decision tree with n! leaf nodes, eg the possible permutations of our input. the dept of out tree is $l \leq 2^d$ which is equal to $\lceil log\ l \rceil \leq d$ Therefor for any decision tree for any algorithm that sorts by comparison of keys has a dept of at least $\lceil log\ n! \rceil$ So the number of comparisons needed to sort in the worst case is at least $\lceil log\ n! \rceil$
using eq 1.9 er get that $log\ n! \geq n\ log\ n - 1.5n$
The above is very close to merge sort ans this tells us that merge sort is very close to being optimal in the worst case.

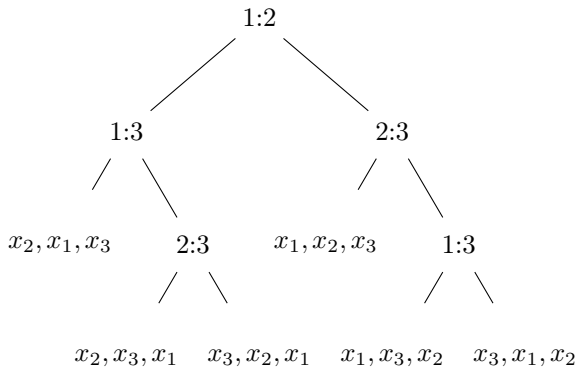


Figure 1.1: **Figure 2.11** Decision tree for a sorting algorithm, $n = 3$.

Lower bound for average behavior

This can be computed by using the external path length denoted epl, for a 2 tree epl is minimized if all leaf nodes are on most two adjacent levels. if they are not we can make surgery on the tree so all nodes are on at most 2 levels. if all leafs are on at most 2 level we get that This gives us that the minium average path lenght is $\lfloor lgl \rfloor + \epsilon$
Eg, on the average case no algorithm can substantially better then merge sort or quick sort on the average case.

9. Adversary arguments – technique, examples.

Introduction

The goal with Adversary arguments is to build the "worst" possible input wrt. run time. An example of this is the problem of min or max

min/max

Scanning over the list and keeping the lowest element we've met so far. n-1 run time.

Least information per comparison. Finding min and max

Max min for 5 element list. is that we want to give as little information as possible and we first assign the final value of a variable when we need to return it and make a relative order while running the problem.

Comparison	x_1		x_2		x_3		x_4		x_5		x_6		Info
	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value	Given
x_1, x_2	**W**	**20**	**L**	**10**	N	*	N	*	N	*	N	*	2
x_1, x_5	**W**	**20**	L	10	**N**	*	N	*	**L**	**5**	N	*	1
x_3, x_4	W	20	L	10	**W**	**15**	**L**	**8**	L	5	N	*	2
x_3, x_6	W	20	L	10	**W**	**15**	L	8	L	5	**L**	**12**	1
x_3, x_1	**WL**	**20**	L	10	**W**	**25**	L	8	L	5	L	12	1
x_2, x_4	WL	20	**WL**	**10**	W	25	**L**	**8**	L	5	L	12	1
x_5, x_6	WL	20	WL	10	W	25	L	8	**WL**	**5**	**L**	**3**	1
x_6, x_4	WL	20	WL	10	W	25	**L**	**2**	WL	5	**WL**	**3**	1

Weight based, finding max and second largest

In the weight based, the adversary assigns value to nodes and follows those nodes. we build a tree and keep pointers to the elements that our root nodes have beat thus far. the the second iteration for the root nodes of our max (log n elements) and find the biggests of those gives us the second largest element.

Comparands	Weights	Winner	New weights	Keys
L[],L[]	w(L[1]) = w(L[2])	L[1]	2,0,1,1,1	20,10,*,*,*
L[],L[]	w(L[1]) >w(L[3])	L[1]	3,0,0,1,1	20,10,15,*,*
L[],L[]	w(L[5]) = w(L[4])	L[5]	3,0,0,0,2	20,10,15,30,40
L[],L[]	w(L[1]) >w(L[5])	L[1]	5,0,0,0,0	41,10,15,30,40

10. Median problem – algorithm and lower bound.

introduction

The problem of finding the median is finding the element in the middle. eg this can be done naively by sorting the list and picking element (n+1)/2

Algorithm

The algorithm we use builds a tree where we have a center node the presumed median and all the keys larger then it is on top and smaller then it is below. and as we need to know the medians relative position compared to all other keys we need to do n-1 comparisons. this is not a great lower bound as it only accounts for the time to verify that it's the median.

lower bound

We can use adversary arguments to show that we can make the algorithm make useless comparisons. this is done by assigning a value to our median but not a key until we have to.
using this method we can make our algorithm do non crucial computations.
The way this works is that we assign values to our nodes based on the relative size to our median. S(maller), L(arger), and N(on).
Using this method we can always put our new node on the bad size compared to the algorithm picked median and making it shift it's input. this can be done until either L or S has (n-1)/2 keys. as we can't put anymore keys on the bad side anymore, but by this time we would have wasted (n-1)/2 non crucial comparisons.
This brings the total lower bound to n-1+(n-1)/2 comparisons, this is again not a great lower bound as adversary arguments doesn't work well for this task.

Selection in worst case linear time.

Today the lower bound has slowly crept up and i currently at around 2n(CLRS) and the best proven method is using the selection algorithm from quick sort.
The way this works is by splitting the input into segments of 5 and sorting using insertion sort. do this again on the input from the first step. This overall gives us a linear runtime.

11. Approximation algorithms

introduction

Many problems in NP are quite significant so we can't abandon them, It may be fine with limited n, or in other cases huge amounts of computational power. but it's not in all cases we need the optimal solution, and the near optimal solution can often be found in linear polynomial time.

In comparison with np-complete

While NP complete problems may take 2^n we can approximate them much faster. this approximation will be within some margin of error.
An example of this is vertex cover
To get a vertex cover we may have to try all possible 2^n markings to test which one has the fewest nodes.
Our approximation algorithm works by

```
approx-Vertex-cover(G(V,E))
        C = empty set
        while E != empty
                Select a arbitrary edge in E{u,v}
                C = C union {u,v}
                Remove every incident edge on {u,v}
        return C;
```

Approximation ration

Is the ratio that we are within of our optimal solution.
For our above problem of vertex cover the optimal solution is n
The definitions says that we have to cover every edge in our graph. eg every node has to be connected to at least 1 marked node. our approach has two nodes in C that are adjacent which gives us the upper bound of

$$|C| \leq 2|C^*| \tag{1.3}$$

If follows that that the ratio for maximization is $\frac{C^*}{C}$ and for minimization $\frac{C}{C^*}$ and P cant be 1 and can only be p > 1, else P = NP is true.

Traveling salesman

The traveling salesman problem is based on the triangle inequality. where we can build a MST, and traverse it. and if we meat repeat nodes on the way back from our transverse we ignore them by removing them from the tree. This gives us a run time of $|V|^2$ which is polynomial.
The cost ration compared to the optimal shown be at most 2C, furthermore. by using the Triangle inequality we can show that the $c(H) \leq c(W)$
However if we do not consider the case where the Triangle inequality then here is no TSP algorithm that runs in p time unless N = NP

Greedy

Set-cover

Randomized

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

Other lecture notes.

Here is the text for topics not covered by Introduction To the thery of computation by Micheal Sipser, Combinatorial Optiomization, Christos H. Papadimitriou, and Kenneth Steighlitz
The following content is from the book Computer algorithms by Sara Baase

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).)

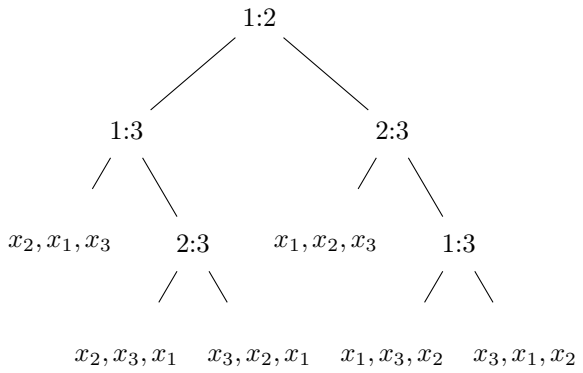


Figure 3.1: **Figure 2.11** Decision tree for a sorting algorithm, $n = 3$.

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. a straightforward induction on d. □

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

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 at least $\lceil logn! \rceil$.

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

We need a lower bound on the average lenght of all paths from the root to a leaf. The *external path length*(epl) of a tree is the sum of the lengths of all paths from the root to a leaf, it will be denoted epl.
A binary tree in which every node has degree 0 or two is called a *2-tree*. our decision trees are 2 trees, so the next two lemmas give us a lower bound on their epl.

Lemma 2.8 Among 2-trees with l leaves the epl is minimized only if all the leaves are on at most two adjacent levels.

Proof. Suppose we have a 2-tree with depth d that has a leaf x at level k, where k$\leq d-2$, we will exhibit a 2-tree with the same number o leaves and lower epl.
Chose a node y at level d-1 that is not a leaf, remove its children, and attach two children to x.(see figure 2.12 for an illustration.) the total number of leaves have not changed. the epl has been decreased by 2d+k, because the path to the children of y and the path to x are no longer counted, and increased by $2(k+1)+d-1 = 2k+d+1$ the sum of the length of the paths to y and the new children of X, there is a net decrease in the epl og $2d+k-(2k+d+1) = d-k-1 > 0$ since $k \leq d-2$ □

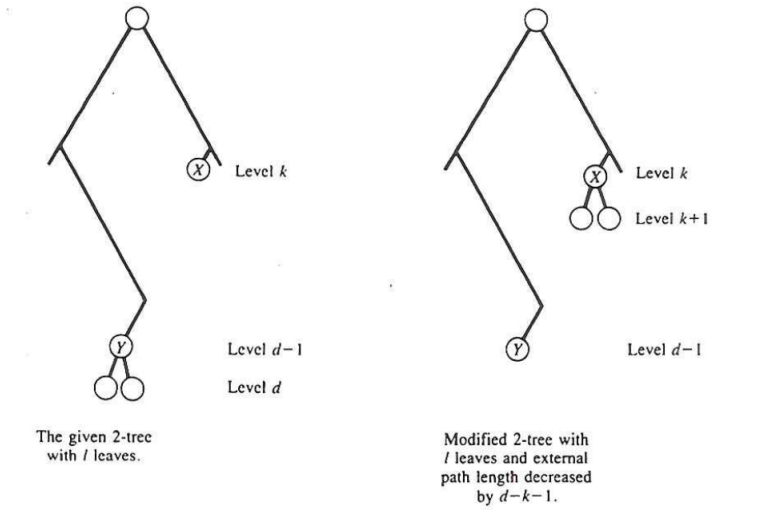


Figure 3.2: **Figure 2.12** Decreasing external path length

Lemma 2.9 The minimum epl for 2 trees with l leaves is $l\lfloor log\ l \rfloor + 2(l - 2^{\lfloor log\ l \rfloor})$

Proof. If l is a power of 2, all the leaves are at level $log\ l$.(This statement depends on both the fact that the tree is a 2 tree and that all leaves are on at most two levels. the reader should verify it) the epl is $l\ log\ l$ which is the value of the expression in the lemma in this case.
If l is not a power of 2 the depth of the tree is $d = \lceil log\ l \rceil$ and all the leaves are at levels d-1 and d. the sum of the path lengths(for all leaves) down to level d-1 is $l(d-1)$. For each leag at level d. 1 must be added to get the total epl. the number of leaves at level d is $2(l-2^{d-1})$, since for each node at level d-1 that is not a leaf there are two leaves at level d. (see figure 2.12) this the sum is $l(d-1)+2(l-2^{d-1}) = l\lfloor log\ l \rfloor + 2(l - 2^{\lfloor log\ l \rfloor}$. □

Lemma 2.10 The average path lenght in 2-tree with l leaves is at least $\lfloor log\ l \rfloor$.

Proof. The minimun average path lengh is

$$\frac{l\lfloor log\ l \rfloor + 2(l - 2^{\lfloor log\ l \rfloor})}{l} = \lfloor log\ l \rfloor + \epsilon \tag{3.5}$$

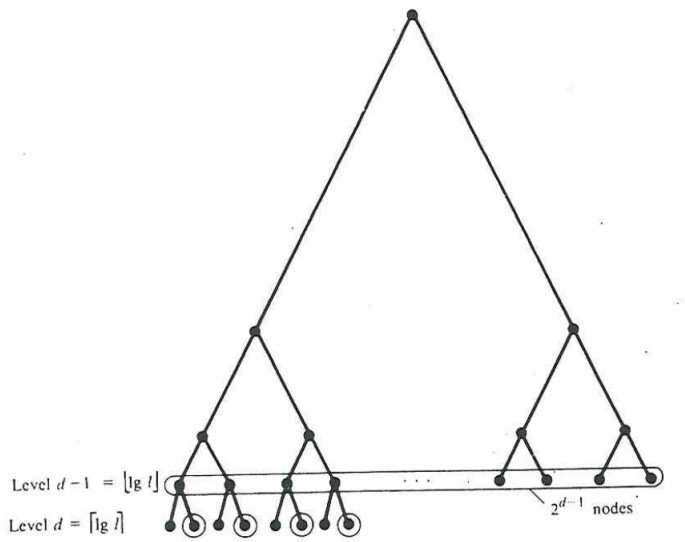


Figure 3.3: **Figure 2.13** Computing external path lengh for lemma 2.9, $l = 2^{d-1}$ + the number of nodes at level d-1 that are not leaves

where $0 \leq \epsilon < 1$ since $l - 2^{\lfloor log\ l \rfloor}$ is always less than l/2 □

Theorem 2.11 The average number of comparisons done by an algorithm to sort n items by comparisons of keys is at least $\lfloor log\ n! \rfloor \approx \lfloor n\ log\ n - 1.5n \rfloor$.
Thus no algorithm can do substantially better then quick sort and merge sort on the average

3.1 - Introduction

3.1.1 The selection problem

Suppose L is an array containing n keys from some linearly ordered set, and let k be an integer such that$1 \leq k \leq n$. The selection problem is the problem of finding the smallest key in L, as with most of the sorting algorithms we've studied, we will assume that the only operations that may be preformed on the keys are comparisons of pairs of keys(as well as copying or moving keys).
In chapter 1 we solved the selection problem for the case k = n, for that problem is simply to find the larges key, we consider a straightforward algorithm that did n-1 key comparisons. and we proved that no algorithm could do fewer. the dual case for k = 1, that is finding the smallest key, can be solved similarly. another very common instance of the selection problem is the case where $k = \lceil n/2 \rceil$, that is where we want to find the middle, or median element
Of course the selection problem can be solved in general by sorting L: then L[k] would be the answer, sorting required $\theta(n\ log\ n)$ key comparisons, and we have just observed that for some values of k, the selection problem can be solved in linear time, finding the median seems, intuitively, to be the hardest instance of the selection problem, can we find the median in linear time? or can we establish a lower bound for the median finding that is more than linear, maybe $\theta(n\ log\ n)$ we will answer these questions in this chapter.

3.2.1 Lower bounds

So far we have used the decision tree as our main technique to establish lower bounds, recall that the internal nodes of the decision tree for an algorithm represents the comparisons the algorithm performs, and that the leaves represent the outputs.(For the search problem in section 1.5, the internal nodes also represented outputs) The number of comparisons done in the worst case is the depth of the tree: the depth is at least $\lceil log\ l \rceil$ where l is the number of leaves.
In chapter 1 we used decision trees to get the (worst case) lower bound of $\lfloor log\ l \rfloor - 1$ for the search, so a decision tree argument gave us the best possible lower bond. in chapter 2 we used decision trees

to get a lower bound of $\lceil log\ n! \rceil$, or roughly $\lceil n\ log\ n - 1.5n \rceil$ for sorting. there are algorithms whose performance is very close to this lower bound so once again a decision tree argument gave us a very strong result. However, decision tree arguments do not work very well for the selection problem.
A decision tree for the selection problem must have at least n leaves because any of the n keys in the list may be the output. i.e the k'th smallest. thus we conclude that the dept of the tree (and the number of comparisons done in the worst case) is at least $log\ n$). But this is not a good lower bound. we already know that even the easy case of finding the largest key requires at least n-1 comparisons, what is wrong with the decision tree argument ? in a decision tree for an algorithm that finds the largest key, some output appear at more then one leaf, and there will in fact be more than n leaves, to see this, draw the decision tree for FindMax(Algorithm 1.3) with n = 4 leaves. the decision tree argument fails to give a good lower bound because we do not have an easy way to determine how many leaves will contain duplicates of a particular outcome. instead of a decision tree we will use a technique called adversary argument to establish a better low bounds for the selection problem.
Suppose you're playing a guessing game with a friend, you are to pick a date (a month and a day) and the friend will try to guess the date by asking yes/no questions. you want to force your friend to ask a many questions as possible. if the first question is "is it in the winter?" and you are a good adversary, you will answer no. because there are more dates in the tree other seasons.to the question "is the first letter of the month's name in the first half of the alphabet=" you should answer "yes", but is this cheating? you did not really pick a date at all. in fact you will next pick a specific month and day until the need for consistency in your answers pin you down. this may not be a friendly way to play a guessing game,but it is just right for finding lower bounds for the behavior of an algorithm.
Suppose we have an algorithm that we think is efficient. imagine an adversary who wants to prove otherwise. at each point in the algorithm where a decision(a key comparison for example) is made. the adversary tells us the result of the decision. the adversary chooses it's answers to try and force the algorithm as gradually construction a "bad" input for the algorithm while it answers the questions. the only constraint on the adversary's answers is that they must be internally consistent, there must be some input for the problem for which it's answers would be correct. if the adversary can force the algorithm to preform f(n) steps, then f(n) is a lower bound for the number of steps in the worst case.
We want to find a lower bound on the complexity of a problem, not just a particular algorithm. therefor when we use adversary arguments, we will assume that the algorithm is any algorithms whatsoever from the class being studied. just as we did with the decision tree argument. to get a good lower bound we need to construct a clever adversary that can thwart any algorithm.
In the test of this chapter we present algorithms for selection problems and adversary arguments for lower bounds for several cases, including the median, in most of the algorithms and arguments we will use the terminology of contents, or tournaments to describe the result of comparisons, the comparand that is found to be larger will be called the winner the other will be the loser.

3.2 - Finding min and max

Throughout this section we will use the names max and min to refer to the largest and smallest keys, respectively, in a list of n keys.
We can find max and min by using algorithm 1.3 to find max, eliminating max from the list. and then using the appropriate variant of the algorithm to find min among the remaining n-1 keys. This max and min can be found b doing n-1+n-2 or 2n-3 comparisons. This is not optimal, although we know (from chapter 1) that n-1 key comparisons are needed to find min and max independently. when finding both some of the work can be "shared" exercise 1.12 asks for an algorithm to find max and min with only about $\frac{3n}{2}$ key comparisons. The solution (For even n) is to pair up the keys and do n/2 comparisons, then find the largest of the winners and the smallest of the losers. (If n is odd, the last key may have to be considered among the winners and the losers). In this section we give an adversary argument to show that this solution is optimal. Specifically we will prove:

Theorem 3.1 Any algorithm to find max an min of n keys by comparisons of keys must do at least $\frac{3n}{2} - 2$ key comparisons in the worst case.

To establish the lower bound we may assume that the keys are distinct. to know that a key x is max and that a key y is min. an algorithm must know that every key other then x has lost some comparison and that every other than y has won some comparison. if we count each win as one unit of information, and each loos as one uni of information then an algorithm must have (at least) 2n-2 units

of information to be sure of giving the correct answer. we give a strategy for an adversary to use in responding to the comparisons so that it gives away as few units of new information as possible with each comparison. Imagine the adversary construction a specific input list as it responds to the algorithm's comparisons.
We denote the status of each key at any time during the course of the algorithms as follows:

Key status	Meaning
W	Has won at least one comparison and never lost
L	Has lost at least one comparison and never won
WL	Has won and lost at least one comparison
N	Has not yet participated in a comparison

The adversary is described in table 3.1 the main point is that. except in the case where both keys have not yet been in any comparisons. the adversary can give a response that provides most at most one unit of information.we need to verify that if the adversary follows these rules, it's replies are consistent with some input.then we need to show that this strategy forces any algorithm to do as many comparisons as the theorem claims.
Observe that in all cases in Table 3.1 except that last, either the key chosen by the adversary as the winner has not yet lost any comparisons, or the key chosen as the loser has not yet won any. consider the first possibility. suppose that the algorithms compares x and y, that the adversary chooses x as the winner, and that x has not yet lost any comparison. Even if the value already assigned by the adversary o x is smaller then the value it has assigned to y. the adversary can change's value to make it bet y without contradicting any of he responses it gave earlier. the other situation where the key chosen as the loser has never won, can be handled similarly – by reducing the value of the key if necessary. So the adversary can construct an input consistent with the rules for responding to the algorithms comparisons This is illustrated in the following example. **Table 3.1**

The adversary strategy for the min and max problem.

[h]

Status of keys x and y Compared by an algorithm	Adversary argument	New status	Units of new information
N,N	x>y	W,L	2
W,N or WL, N	x>y	W,L or WL, L	1
L,N	x <y	L,W	1
W,W	x >y	W, WL	1
L,L	x>y	WL,W	1
W,L, or WL,L og W, WL	x>y	no change	0
WL, WL	consistent with assigned values	no change	0

Example 3.1 Constructing an input using the adversary's rules

The first column in table 3.2 shows a sequence of comparisons that might be carried out by some algorithm. the remaining columns show status and values assigned to the keys by the adversary. (keys that have not yet been assigned a value are denoted by asterisk.) each row after the first contains only the entries relevant for the current comparison.Note that when x_3 and x_1 are compared (in the fifth comparison), the adversary increase the values of x_3 because x_3 is suppose to win. later the adversary changes the values of x6 and x4 consistent with it's rules, After the first five comparisons every key except x_3 has lost at least once, so x_3 is max. after the last comparison x_4 is the only key that has never won. so it is min, in this example the algorithm did eight comparisons, the worst case lower bound for size keys.(still to be proved) is $3/2\dot{6} - 2 = 7$

Table 3.2
An Exsample of the adversary strategy

Comparison	x_1		x_2		x_3		x_4		x_5		x_6		Info
	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value	Status	Value	Given
x_1, x_2	**W**	**20**	**L**	**10**	N	*	N	*	N	*	N	*	2
x_1, x_5	**W**	**20**	L	10	**N**	*	N	*	**L**	**5**	N	*	1
x_3, x_4	W	20	L	10	**W**	**15**	**L**	**8**	L	5	N	*	2
x_3, x_6	W	20	L	10	**W**	**15**	L	8	L	5	**L**	**12**	1
x_3, x_1	**WL**	**20**	L	10	**W**	**25**	L	8	L	5	L	12	1
x_2, x_4	WL	20	**WL**	**10**	W	25	**L**	**8**	L	5	L	12	1
x_5, x_6	WL	20	WL	10	W	25	L	8	**WL**	**5**	**L**	**3**	1
x_6, x_4	WL	20	WL	10	W	25	**L**	**2**	WL	5	**WL**	**3**	1

Proof. To complete the proof of theorem 3.1 we need only show that the adversary rules will force any algorithm to do at least $3n/2 - 1$ comparisons to get $2n - 2$ units of information it needs. The only case where an algorithm can get two units of information from one comparison is the case where two keys have not been included in any previous comparisons. suppose for the moment that n is even. An algorithm can do at most $n/2$ comparisons of previously unseen keys. so it can get at most n units of information this way. From each comparison. it gets at most one unit of information. this to get $2n-2$ units of information an algorithm must do at least $n/2+n-2 = 3n/2-2$ comparisons in total. the reader can easily check that for odd n, at least $3n/2-3/2$ comparisons are needed.This completes the proof for theorem 3.1 □

3.3 Finding the second largest key

3.3.1 introduction

Throughout this section we will use *max* and *SecondLargest* to refer to the largest and second-largest keys, respectively. for the simplicity in describing the problem and algorithms, we will assume that the keys are distinct.
The second largest key can be found with 2n-3 comparisons by using findmax(algorithm 1.3)twice. but this is not likely to be optimal. we should expect that some of the information discovered by the algorithm while finding max can be used to decrease the number of comparisons preformed in finding second largest. all such keys are discovered while finding max may be ignored during the second pass though the list.(The problem of keeping track of them will be considered later.)
Using algorithm 1.3 on a list with five keys. the result might be as follows:

Comparands	Winner
L[1],L[2]	L[1]
L[1],L[3]	L[1]
L[1],L[4]	L[4]
L[4],L[5]	L[4]

Then max = L[4] and secondlargest is either L[5] or L[1]because both L[2] and L[3] lost to L[1], this only one more comparison is needed to find secondlargest in this example.
It may happen, however that during the first pass though the list to find max we no not obtain any information useful for finding secondlargest. Not necessarily. In the preceding discussion we used a specific algorithm. no algorithm can find max faster then n-1 comparisons, but another may provide more information for eliminating some during doing the second pass though the list. The tournament method, described next provides such information.

3.3.2 The Tournament method

The tournament method is so name because it preforms comparisons in the same way that tournaments are played. keys are paired off and compared in "rounds" in each round after the first one the winners from the preceding rounds are paired off and compared. (if at any round the number of keys are odd one of them simply waits for the next round.) A Tournament can be described by a tree diagram as shown in figure 3.1, each leaf contains a key, and at each subsequent level the parents of each

pair contains the winner. the root will contain the largest key, as in algorithm 1.3, n-1 comparisons are done to find max.
In the process of finding max, every key except max loses in one comparisons, how many lose directly to max? roughly halt the keys in one round will be losers and will not appear in the next round, if n is a power of 2, there are exactly log n rounds, in general the number of rounds is $\lceil log\ n \rceil$ since max in involved in at most one comparison in each round, there are at most $\lceil log\ n \rceil$ keys that lost only to max and this could possibly be second largest. The method of algorithm 1.3 can be used to find the largest of these $\lceil log\ n \rceil$ keys by doing $\lceil log\ n \rceil - 1$ comparisons, thus the tournament finds max and second largest by doing a total of $n + \lceil log\ n \rceil - 2$ comparisons. this is an improvement over our first result of 2n-3. can we do better?

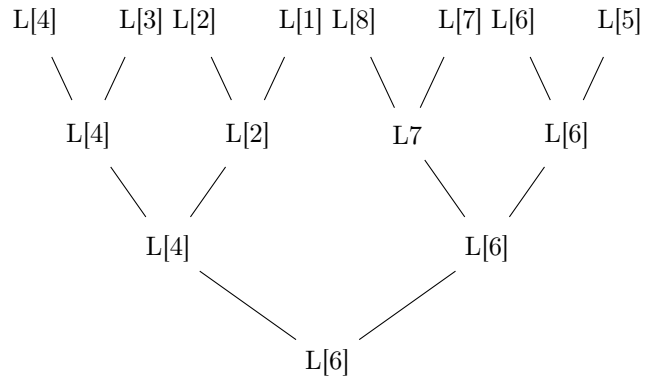


Figure 3.4: **Figure 3,1** An example of a tournament, max = L[6] secondlargest may be L[4],L[5], or L[7]

3.3.3 - An Adversary Lower bound Argument

Both methods we considered for finding the second largest key first found the largest key, this is not wasted effort, any algorithm that finds seconds largest must also find max because to know that a key is the seconds largest one must also that it is not the largest, that is it most have lone in one comparison. the winner of the comparison in which second largest looses must of course be max, this argument gives a lower bound on the number of comparisons needed to find the seconds largest namely n-1 because we already know that n-1 comparisons are needed for max, but one would expect that this lower bound could be improved because an algorithm to find secondslargest should have to do more work than an algorithm to find max. we will prove the following theorem which has as a corollary that the tournament sort is optimal.

Theorem 3.2 Any algorithm (That works by comparing keys) to find the second largest in a list of n keys most do at least $n + \lceil log\ n \rceil - 2$ comparisons in the worst case.

Proof. For the worst case, we may assume that the keys are distinct. We have already observed that there must be n-1 comparisons with distinct losers, If max was a comparand in $\lceil log\ n \rceil$ of these comparisons, then all but one of the $\lceil log\ n \rceil$ keys that lost to max must lose again for second largest to be correctly determined. Then a total of at least $n + \lceil log\ n \rceil - 2$ comparisons would be done. Therefor we will show that there is a an adversary strategy that can force any algorithm that finds secondlargest to compare max to $\lceil log\ n \rceil$ distinct keys. □

The adversary assigns a "weight" w(x) to each key in the list, initially w(x) = 1 for all x, when the algorithm compares two keys, x and y, the adversary determines it's reply and modifies the weights as follows.

Case	Adversary reply	Updating of the weighths
w(x)>w(y)	x>y	w(x) := w(x)+w(y); w(y) = 0
w(x) = w(y) >0	x>y	w(x) := w(x)+w(y); w(y) = 0
w(y)>w(x)	y >x	w(y) := w(x)+w(y); w(x) = 0
w(x) = w(y) = 0	Consistent with previus replies	No change.

We need to verify that if the adversary follows this strategy, it's replied are consistent with some input. and that max will be compared to at least $\lceil log\ n \rceil$ distinct keys.
These conclusions follow from a sequence of easy observations.

1. A key has lost a comparison if and only if it's weight is zero.
2. In the first tree cases the key chosen as the winner has a non zero weigh, so it has not yet lost. the adversary can give t an arbitrarily high value to make sure it wins without contradicting any of it's earlier replies.
3. the sum of the weights is always n. this is true initially and the sum is preserved bu the updating of the weights.
4. When the algorithm stops only one key can have a nonzero weight. otherwise there would be at least two keys that never lost a comparison and the adversary could choose values to make the algorithms choice of second largest incorrect.

Lemma 3.3 Let x be the key that has nonzero wright when the algorithm stops. then x = max, and x has directly won against at least $\lceil log\ n \rceil$ distinct keys.

Proof. By fact 1, 3, and 4, when the algorithm stops. w(x) = n let $w_k = w(x)$ just after the kth comparison won by x against the previously undefeated key. Then by the adversary's rules.

$$w_k \leq 2w_{k-1} \tag{3.6}$$

Now let K be the number of comparisons x wins against previously undefeated keys.
Then

$$n = w_K \leq 2^K_{w()} = 2^K \tag{3.7}$$

Thus $K \geq log\ n$ and since K is in integer, $K \geq \lceil log\ n \rceil$. The K keys counted here are of course distinct, since once beaten by x a key is no longer "previusly undefeated" and will not be counted again, (even if an algorithm follishly compares it to x again.) □

Another way of looking at the adversary's activity is that it builds trees to represent the orderings relations between the keys. if z is the parents of y, then x beat y in comparison. figure 3.2 shows an example. the adversary combines two trees only when their roots are compared, if the algorithm compares non roots, no change is made in the trees, the weight of a key is simply the number of nodes in that key's tree, if it is a root and other zero otherwise.

Example 3.2 The adversary strategy in action
To illustrate the adversary's action and show how it's decision corresponds to the step-by-step construction of an input, we show an example for n = 5. keys in the list that have not yet been specified are denoted by asterisk. this initially the keys are *,*,*,*,*, note that values assigned to some keys may be changed at a later time. see table 3.3 which shows just the first few comparisons, those that find the max, but not enough to find secondlargest. the weights and the values assigned to the keys will not be changed by any subsequent comparisons.
Diagrams on page 132 TODO
Table 3.3
An Example of the adversary strategy

Comparands	Weights	Winner	New weights	Keys
L[],L[]	w(L[1]) = w(L[2])	L[1]	2,0,1,1,1	20,10,*,*,*
L[],L[]	w(L[1]) >w(L[3])	L[1]	3,0,0,1,1	20,10,15,*,*
L[],L[]	w(L[5]) = w(L[4])	L[5]	3,0,0,0,2	20,10,15,30,40
L[],L[]	w(L[1]) >w(L[5])	L[1]	5,0,0,0,0	41,10,15,30,40

3.3.4 Implementation of the Tournament method for finding max and secondlargest.

To conduct the tournament to find max we need a way to keep track of the winners in each round, this can be done by using an extra array of pointers or by careful indexing if the keys may be moved so that the winner is always placed in say the higher-indexed cell of the two being compared, we leave the choice and details to the reader.
After max has been found by the tournament, only those keys that lose to it are to be compared to find secondlargest. How can we keep track of the elements that lose to max when we do not know in advance which key is max? one way is to maintain linked lists of keys that lose to each undefeated key. This can be done by allocating an array for links indexed to correspond to the keys. initially all links are zero. after each comparison in the tournament, the key that lost would be added to the winner's loser list.(at the beginning of the list) see fig 3.3 for an example. when the tournament in complete and max has been found it is easy to find secondlargest by traversing max's loser list.

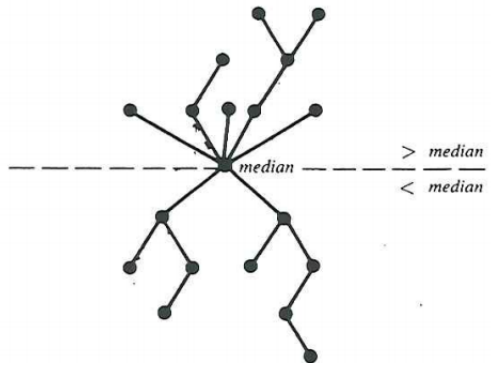
Time and space

The tournament method for finding max and secondlargest uses $\theta(n)$ extra space for links. the running time of the algorithm is in $\theta(n + \lceil log\ n \rceil - 2) = \theta(n)$ since the number of operations for the links is roughly proportional to the number of comparisons done.
We can find the largest and second largest keys in a list by using findmax twice, doing 2n-3 comparisons. or we can use the more complicated tournament method. doing $n + \lceil log\ n \rceil - 2$ comparisons at most. which method is better, the results of exercise should be instructive, both algorithms are in $\theta(n)$) since tournament method uses more instructions per comparison while finding max, it may very well be slower. it is also more complicated the main point considering this problem was not to find a algorithm that beats the straightforward in practice, but to illustrate the adversary argument for the lower bound and bu exhilarating both the adversary argument and the tournament algorithm to determine the optimal number of operations.

There are some missing pages here

3.5 A lower bound for finding the median

We are assuming that L is a list of n keys and that n id odd. we will establish a lower bound on the number of key comparisons that must be done by any key-comparison algorithm to find median, the $(n+1)/2th$ key. since we are establishing a lower bound we may without loss of generality, assume that the keys are distinct.
We claim first that to know median an algorithm must know the relation of every other key to the median. That is for every other key x, the algorithm must now that x $>$ median or x $<$ median. in other words it must establish relations as illustrated by the tree in fig 3.6, each node represents a key and each branch represents a comparison. The key at the higher end of the branch is the larger key. suppose there were some key, say y, whose relation to median was not known, (see fig. 3.7(a) for an example) an adversary could change the value of y moving it to the opposite side of the median as in fig 3.7(b) without contradicting the results of any of the comparisons done. then median would not be the median, the algorithms answer would be wrong.
Since there are n nodes in the tree in fig 3.6 there are n.1 branches so at least n.1 comparisons must be done. This is neither a surprise nor an exciting lower bound. we will show that an adversary lower can force an algorithm to do other "useless" comparisons before it preforms the n-1 comparison it needs to establish the tree of figure 3.6.



Definition A comparison involves a key x is a crucial comparison if it is the first comparison where $x > y$, for some $y \geq median$or $x < y$ for some $y \leq median$(This is the comparison that establishes the relation of x to median, note that the definition does not require that the relation of y to median be already known at the time the crucial comparison for x is done.)

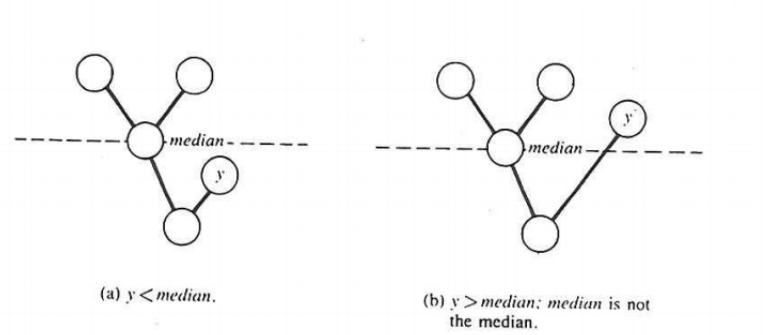


Figure 3.7 An adversary conquers a bad algorithm.

Comparisons of x and y where x > median and y < median are non crucial. we will exhibit an adversary that forces an algorithm to preform such comparisons. the adversary chooses some value (but not a particular key) to be median. it will assign a value to a key when the algorithm first uses that key in a comparison, so long as it can do so the adversary will assign values to new keys involved in a comparison so as to put the keys on the opposite side of the median. The adversary may not assign values larger then the median to more than (n-1)/2 keys. it keeps track of the assignments it has made to be sure not to violate these restrictions. we indicate the status of a key during the running of the algorithm as follows:

L Has been assigned a value larger then median.
S Has been assigned a value smaller then median The adversary strategy is summed up in table
N Has not yet been assigned a value.

3.4 in all cases if there are already (n-1)/2 keys with status S(or L) the adversary ignores the rule in the table and assigns value(s) larger(or smaller) then median to new key(s) when only one key without value remains the adversary assigns the value median to that key. whenever the algorithm compares two keys with status L and L, S and S, or L and S the adversary gives the correct response based on the values already assigned to the keys.
All of the comparisons are describes in table 3.4 are noncrusial, how many can the adversary make any algorithm do? each of these comparisons creates at most one L key. and each creates at most one S key. since the adversray is free to make the indicated assignment until there are (n-1)/2 L keys or (n-1)/2 S keys it can force any algorithm to do at least (n-1)/2 noncrucial comparisons.(since an algortihm could start out by doing (n-1)/2 comparisions involding two N keys this adversary cannot guarantee any more then (n-1)/2 noncrucial comparisons) We can now conclude that the total number of comparisons must at least be n-1(The crucial comparisons)+n-1/2 (the crucial comparisons) we sum up the result in the following theorem.

Theorem 3.4 Any algorithm to find the median of n keys(for odd n) by comparisons of keys must do at least 3n/2-3/2 comparisons in the worst case.

Out adversary was not as clever as it could have been in it's attempt to force an algorithm to do non-

cucial comparisons, in the past several years the lower found for the median problem has crept up the roughly 1.75n-log n, then roughly 1.8n then a little higher. the best lower bound currently known is slightly above 2n for large n, there is still a small gap between the best known and the lower bound and the best known algorithm for finding the median.

Chapter 4

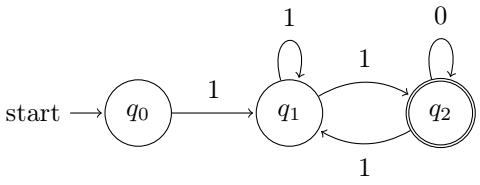
Exercises 2019

Week 1

page 84, question 1.7

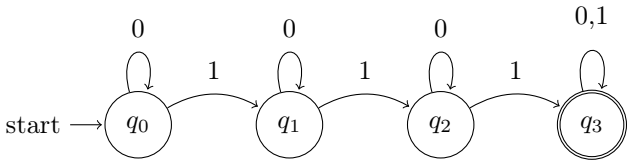
a

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



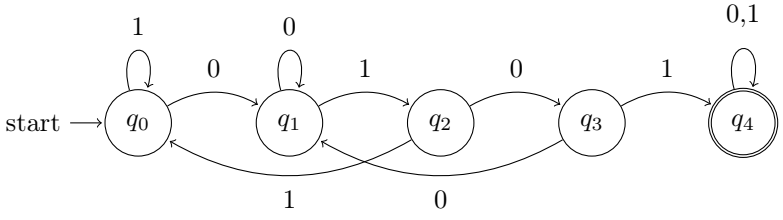
b

w|w contains at least 3 1's



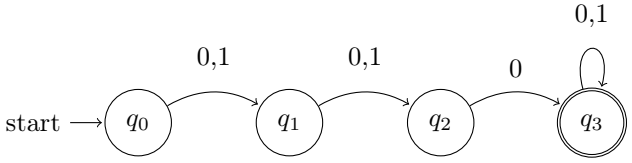
c

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



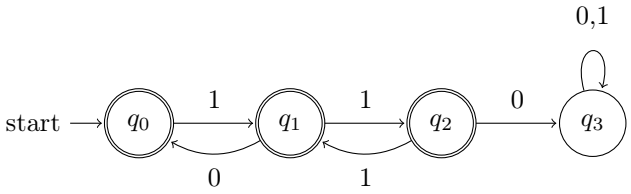
d

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



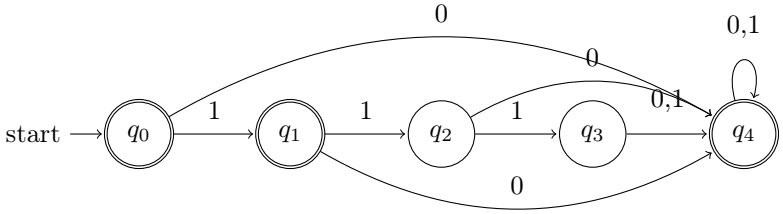
f

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.



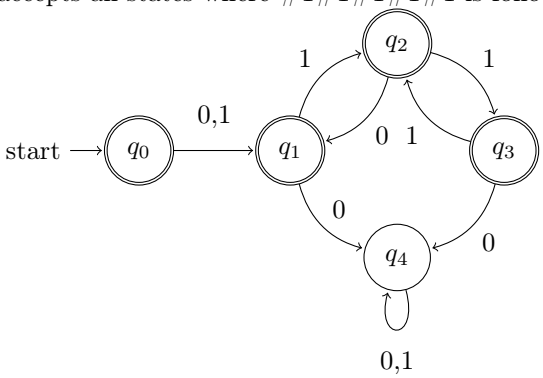
h

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



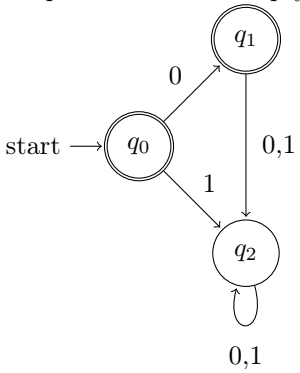
i

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

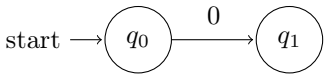
w|w is only the empty string and 0
accepts "0" and the empty string ϵ



page 84, question 1.7

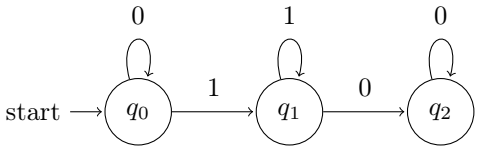
d

The language 0 with two states,



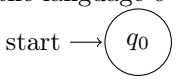
e

the language 0*1*0* with 3 states



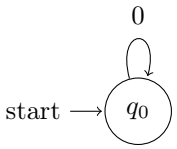
g

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.

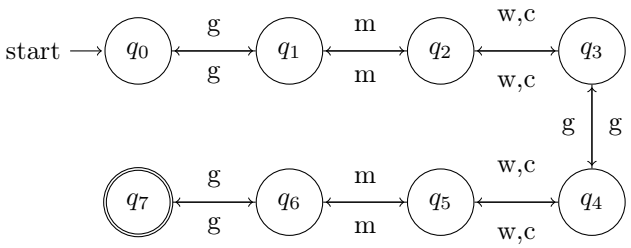
- First you carry the goat to the other side, and go back empty.
- The you ferry the wolf to the other side, and swap with the goat and bring the goat back.
- you then swap the goat with the cabbage and bring it to the other side.
- lastly you head back empty and bring the goat.
- you now have all the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



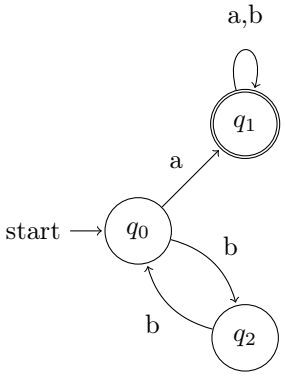
Week 2

page 86, question 1.16

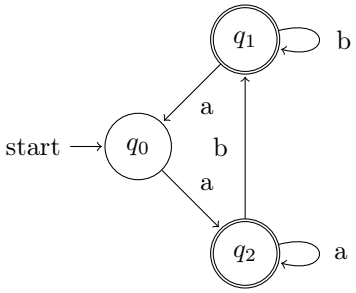
* is zero or more
! is one or more

a

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



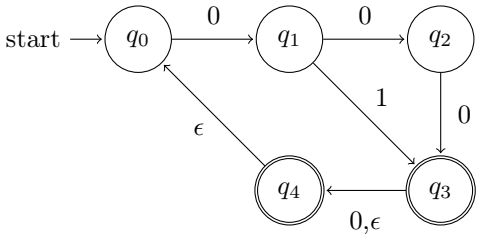
b



page 86, question 1.17

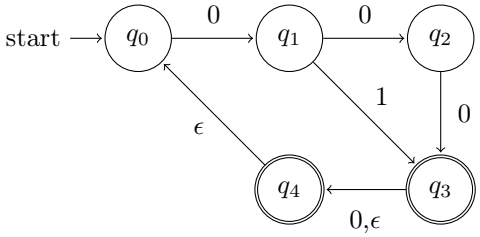
a

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

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

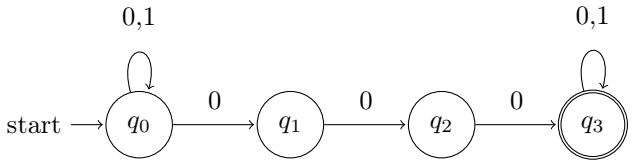
g

$yyyyy$

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, abababbabab,
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

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

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

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

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 flooowing, 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)

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

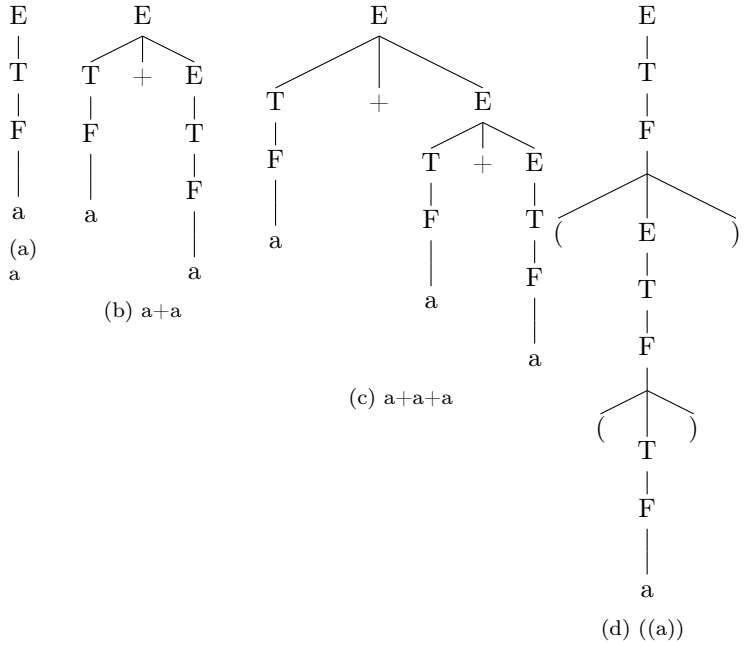


Figure 4.1: 2 Figures side by side

page 154. question 2.4

a

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

b

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

c

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

d

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

e

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

f

$S \to \epsilon$

2.6

Give the CFG that generates the following languages

b

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

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

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.
$S \to FG$
$E \to GE|FE|\epsilon$
$G \to 1G|\epsilon$
$F \to 0F|\epsilon$

2.14

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

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

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

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

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

Concatenation

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

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 $A/B = \{w|wx \in A \text{ for some } x \in B\}$, 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,$B = \{uv | u \in \sum^*, v \in \sum^* 1 \sum^* \text{ and } |u| \geq |v|\}$

a

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

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