

CDM Homework Problems

KLAUS SUTNER

<http://www.cs.cmu.edu/~sutner>

© 2002–2011

Contents

1	Logic	6
1.1	Dual Formulae	6
1.2	Biconditionals	6
2	Computation	8
2.1	LOOP and Primitive Recursion	8
2.2	Shallow Loop Programs	8
2.3	Some Primitive Recursive Functions	9
2.4	Course-of-Value Recursion	10
2.5	The Busy Beaver Function	10
2.6	The Busy Beaver Function (Programs)	11
2.7	Overhead-Free Automata	12
2.8	Write-First Turing Machines	12
2.9	Binary Register Machines	13
2.10	Reduced Turing Machines	14
2.11	Semi-Decidable Sets and Computable Functions	15
2.12	Graphs of Computable Functions	16
2.13	Vertex Cover and SAT	16
2.14	Easy Satisfiability	17
2.15	Partition	17
2.16	Kolmogorov Complexity	18
2.17	Uninspired Sets	18
2.18	Expressiveness of FOL	19
3	Induction, Iteration	20
3.1	Spreading Negativity	20
3.2	Hereditarily Finite Sets	20
3.3	Inverse Sequences	21
3.4	Mystery Recursion	22

3.5	Nested Recursion	23
3.6	Powers of Two	24
3.7	Collapsing Transformations	25
3.8	Reversal and Palindromes	25
3.9	Lookup Tables and Iteration	26
3.10	Iteration and Composites	26
3.11	Fast Exponentiation	27
3.12	Greatest Common Divisor	28
3.13	Binary Square Roots	29
3.14	The Josephus Problem	29
3.15	Thue and Shuffle	30
3.16	Fibonacci Words	31
3.17	Tournaments and Kings	32
3.18	Tournaments and Fairness	33
3.19	Ducci Sequences	33
3.20	The DAZS Operator	34
3.21	UnCollatz	35
3.22	Son of Collatz	36
3.23	Floyd's Cycle Detection Trick	37
3.24	Floyd and Teleportation	38
3.25	Schröder-Bernstein	39
3.26	Cardinalities	39
4	Combinatorics	40
4.1	Flattening Multisets	40
4.2	Pruning Labels	41
4.3	Power Sums	42
4.4	3,5,7-Ordered Sequences	43
4.5	A Permutation	43
4.6	Kraft's Inequality	44
4.7	Non-Decreasing Functions	45
4.8	Tic-Tac-Toe	45
4.9	Generating Permutations	46
4.10	Manhattan Paths	46
4.11	Chomp	47
4.12	Necklaces	48
4.13	Flipping Pebbles	49
4.14	Shuffle	50

4.15	The 15 Puzzle	51
4.16	Reversible Gates	52
4.17	Hamiltonian Sequences	52
4.18	Boolean Circuits	53
4.19	Four Fours	54
4.20	Rigid Words	55
4.21	Keane Products	55
5	Finite State Machines	57
5.1	Primitive Words	57
5.2	Dense Languages	57
5.3	Well-Ordered Languages	58
5.4	Word Periods	59
5.5	Combinatorics on Words	59
5.6	Converting Regular Expressions	60
5.7	Regularity and Language Operations	60
5.8	Regularity and Palindromes	61
5.9	Word Binomials	61
5.10	Constrained Quotients	62
5.11	Word Shuffle	62
5.12	Shuffle Language	63
5.13	Frontiers	64
5.14	Frontier Automata	65
5.15	A Groupoid	66
5.16	A Quadratic Language Equation	67
5.17	Forbidden Factors	67
5.18	Subwords and Halfs	68
5.19	The Un-Equal Language	68
5.20	Representations of Regular Languages	70
5.21	Primitivity and Minimality	70
5.22	Definite Automata	71
5.23	Right Quotients	71
5.24	Direct Languages	72
5.25	Prefix Languages	72
5.26	Recognizing Suffixes	72
5.27	Local Languages	73
5.28	D_4 Recognizer	73
5.29	Recognizing Permutations	74

5.30	Balance and Majority	74
5.31	State Merging	75
5.32	Minimal Automata for Finite Languages	75
5.33	Fast Equivalence Testing	76
5.34	Counting Minimal 1-Letter DFAs	77
5.35	Divisibility In Reverse Binary	77
5.36	Forward State Merging	78
5.37	Determinization	78
5.38	Determinization and Blowup	79
5.39	Universal Finite Automata	79
5.40	The Dyck Language	80
5.41	Pumping	80
5.42	DFAs versus Regular Expressions	81
5.43	Hard Regular Expressions	81
5.44	ω -Regular Languages	82
5.45	Acceptance for Büchi Automata	82
5.46	Determinization of Büchi Automata	83
5.47	Counting Letters	85
5.48	Solving Language Equations	86
5.49	Multiplicity	86
5.50	Small NFAs	86
5.51	Fractional Languages	87
6	Cellular Automata	88
6.1	Affine Shift Registers	88
6.2	Linear Congruential Generators	89
6.3	All-Ones	89
6.4	Two Simple Cellular Automata	90
6.5	Additive Cellular Automata	90
6.6	Affine Cellular Automata	91
6.7	Hybrid Cellular Automata	92
6.8	Additive ECA 90	92
6.9	Building Reversible Cellular Automata	94
6.10	Counting Boolean Functions	96
6.11	Counting Local Maps	97
6.12	Analyzing Simple Elementary Cellular Automata	97
7	Algebra	99
7.1	Modular Multiplication	99

7.2	An Associative Operation	100
7.3	Generating Permutations and Functions	100
7.4	Matrices and Words	101
7.5	Boolean Rings	102
7.6	Boolean Algebras	102
7.7	Boolean Algebras without Times	103
7.8	A Transformation Semigroup	104
7.9	Floyd goes Algebraic	104
7.10	Polynomial Equations Mod 2	105
7.11	Chessboards and Lights	106
7.12	Shrinking Dimension	107
7.13	Building A Finite Field	108
7.14	Moving Cubes	108
7.15	Characteristic 2	109
7.16	Redundant Field Representations	110

Chapter 1

Logic

1.1 Dual Formulae

Background

For this problem let us only consider Boolean connectives \neg , \vee and \wedge . For any propositional formula φ its *dual* φ^{op} to be obtained from φ by interchanging \wedge and \vee , as well as \perp and \top . For example, $(p \vee \neg q)^{\text{op}} = p \wedge \neg q$. Similarly, for a truth assignment v , define $v^{\text{op}}(p) = \neg v(p)$.

Task

- A. Show that the dual is an involution: $(\varphi^{\text{op}})^{\text{op}} = \varphi$.
- B. Show that $\text{eval}(\varphi^{\text{op}}, v) = \neg \text{eval}(\varphi, v^{\text{op}})$.
- C. Show that two formulae are equivalent, if, and only if, their duals are equivalent.
- D. Show that φ is a tautology if, and only if, φ^{op} is a contradiction.

Comment These properties are more or less obvious, but try to come up with clear, elegant proofs.

1.2 Biconditionals

Background

Suppose we have propositional variables p_1, \dots, p_n . Define formulae $\varphi_1 = p_1$ and

$$\varphi_{i+1} = \varphi_i \leftrightarrow p_{i+1}.$$

For example, $\varphi_3 = (p_1 \leftrightarrow p_2) \leftrightarrow p_3$.

Task

- A. Show that the biconditional is associative and commutative in the sense that the corresponding rearrangements do not affect truth values:

$$\begin{aligned}\varphi \leftrightarrow \psi &\equiv \psi \leftrightarrow \varphi \\ (\varphi \leftrightarrow \psi) \leftrightarrow \rho &\equiv \varphi \leftrightarrow (\psi \leftrightarrow \rho)\end{aligned}$$

- B. Give a simple characterization for the truth of φ_n in terms of the truth values of the propositional variables p_i .

Comment The simpler your characterization in part B. the better.

Chapter 2

Computation

2.1 LOOP and Primitive Recursion

Background

We have seen in class that primitive recursion is one way of defining a class of “easily” computable arithmetic functions, LOOP programs are another. At first glance, there seems to be little connection between the two approaches, yet they define exactly the same class of functions.

Task

- A. Show that every primitive recursive function is LOOP-computable.
- B. Show that every LOOP-computable function is primitive recursive.

Comment

The first part is by induction on the build-up of the primitive recursive function (recall our little p.r. programming language). You can think of this as showing how to express a program in one language by an equivalent program in another. This is not too hard.

Part 2 is a bit tricky. Again, this is a problem of simulating one program by another. However, this time the source program involves variables, so you need to deal with the change of the values of all the variables during execution. To give a super-precise proof you would have to start with a careful definition of $\llbracket P \rrbracket$, the semantics of a loop program P , and then show that all you need to do this is p.r. Even if you are a rather relaxed about semantics it’s probably a good idea to use induction on the depth of the loop program. Needless to say, primitive recursion is used to deal with loops. Try to come up with a solid, elegant argument.

2.2 Shallow Loop Programs

Background

Recall the LOOP programming language from class:

constants	$0 \in \mathbb{N}$
variables	x, y, z, \dots ranging over \mathbb{N}
operations	increment $x++$
assignments	$x = 0$ and $x = y$
sequential composition	$P; Q$
control	do $x : P$ od

It is tempting to measure the complexity of a LOOP program in terms of the nesting depth of the loops in the program. E.g., a program without any loops has depth 0, the LOOP program for addition has depth 1, the multiplication program has depth 2 and so on.

It is not clear that anything interesting happens at low loop depth, but as it turns out things spin out of control rather quickly. More precisely, depth 0 is not particularly interesting, but even depth 1 is already surprisingly complicated. Don't even think about depth 2, things get out of hand there.

Task

- Characterize the functions computable by a LOOP program of depth 0.
- Characterize the functions computable by a LOOP program of depth 1.

Comment

This means that you have to come up with a nice, concise description: a function f is LOOP depth 0 if, and only if, blah-di-blah-blah-di-blah. Of course, you also have to prove that your description works. So you might claim (wrongly!) that “a function is loop depth 0 if, and only if, it is constant or the identity”. Be careful with the proof, so you can detect errors in your characterization.

2.3 Some Primitive Recursive Functions

Background

To show that a function f is primitive recursive it is usually best to argue in terms of closure properties: such and such functions and relations are p.r., and f can be defined in from these using such and such operations, hence f must be p.r., too. For example, we have shown in class that p.r. functions are closed under bounded search.

Task

- Show that the relation $d(x, y)$: “ x divides y ” is primitive recursive.
- Show that the relation $p(x)$: “ x is prime” is primitive recursive.
- Show that the function that maps x to the least prime larger than x is primitive recursive.
- Show that the function $n \mapsto p_n$ where p_n is the n th prime is primitive recursive.

Make sure to lean heavily on the results from lecture, writing down an explicit p.r. definition is far too tedious for this.

2.4 Course-of-Value Recursion

Background

By definition, primitive recursive functions are closed under “classical recursion”:

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

where only the previous value $f(x, y)$ of the function is available in the computation for $f(x + 1, y)$. However, on occasion it is necessary to use all previous values of a function to define the next value (e.g. think about the counting argument for binary trees). This is known as course-of-value recursion.

Informally, we would like to be able to define

$$\begin{aligned}f(0, y) &= g(y) \\ f(x + 1, y) &= h(x, f(x, y), f(x - 1, y), \dots, f(0, y), y)\end{aligned}$$

where g and h are given primitive recursive functions. Formally, this is a bit tricky since h has variable number of arguments, so this definition does not quite work for primitive recursive functions in the form stated.

Task

- A. Explain how fix up the last definition so we can apply course-of-value recursion to primitive recursive functions.
- B. Show that the primitive recursive functions are closed under course-of-value recursion: in the amended definition, if g and h are primitive recursive then so is f .

Comment You probably will want to lean heavily on the results from lecture, writing down an explicit p.r. definition for f is far too tedious.

2.5 The Busy Beaver Function

Background

Let $\beta(n)$ be the Busy Beaver function from class. We mentioned that β is not computable, but have given no proof so far. At the heart of the issue lurks the Halting problem, as usual. However, in the case of β it is easier to show that the function grows faster than any computable function.

Task

- A. Assume $f : \mathbb{N} \rightarrow \mathbb{N}$ is a strictly increasing computable function. Show that for some sufficiently large x we must have $f(x) < \beta(x)$. Note that f is computable by some Turing machine with a fixed number of states.
- B. Conclude that β is not computable.

- C. Herr Prof. Dr. Wurzelbrunft is selling a device on eBay called HaltingBlackBox™ that supposedly solves the Halting Problem. Explain how Wurzelbrunft’s gizmo could be used to compute β . Also explain why it is not a good idea to bid on Wurzelbrunft’s device.
- D. Show that $\beta(5) = 4098$.

Comment

Relax, part D is a joke, you don’t need to do this.

Pinning down $\beta(5)$ would be an excellent project for this class. Look at the Marxen-Buntrock example and think hard about what is needed to realize that this machine is not in a loop. Likewise, think about how you could eliminate machines that are “looping” (i.e., machines that will never halt and just write more and more 1’s on the tape).

2.6 The Busy Beaver Function (Programs)

Background

The Busy Beaver function β is another famous example of a non-computable function. For our purposes, let’s define $\beta(n)$ as follows. Consider all n -bit programs $p \in \mathbf{2}^n$. When we execute such a program in our environment U , we either obtain some output $U(p) = 000\dots00$ or the computation fails to halt and there is no output. Let $\beta(n)$ be the maximum length of any output $U(p)$ consisting only of a finite number of 0’s (and p must halt at some point). Thus, we do not consider programs that output an infinite stream of 0’s or such like, nor do we consider programs that print symbols other than 0’s.

It is intuitively clear that β is not computable: we have no way to eliminate the non-halting programs $p \in \mathbf{2}^n$ from the competition. Alas, it’s not so easy to come up with a clean proof. One line of reasoning is somewhat similar to the argument that shows that the Ackermann function is not primitive recursive: one shows that β grows faster than any computable function.

Task

1. Assume $f : \mathbb{N} \rightarrow \mathbb{N}$ is a strictly increasing computable function. Show that for some sufficiently large x we must have $f(x) < \beta(x)$.
2. Conclude that β is not computable.
3. Suppose Prof. Dr. Blasius Wurzelbrunft is offering a device called HaltingBlackBox™ at his website. Wurzelbrunft claims that his device solves the Halting Problem: given a program p it will decide whether p halts when executed in environment U . Explain how Wurzelbrunft’s gizmo could be used to compute β .

Comment

The BB function is closely related to Kolmogorov-Chaitin program-size complexity: the program `for(i = 0; i < n; i++) print(0);`

has size about $\log n + c$ bits and certainly prints n 0’s. But there may be a much smaller program with the same output: we could replace n by a sub-program p_n of size $C(n)$.

Also note that a BB program p (one that prints the most 0's in its class of n -bit programs) must essentially be incompressible in the Kolmogorov-Chaitin sense. Otherwise we could replace p by a program q than contains \widehat{p} and works like so: compute p from \widehat{p} , and then run p twice. If n is sufficiently large, the size of q can be made to be n (pad with nonsense bits if necessary), but q produces output twice as long as p 's output.

The classical BB function uses (Turing) machines rather than programs and the limitation is on the number of states of the machines. The big surprise here is that even for tiny values of n such as 5 things spin out of control.

2.7 Overhead-Free Automata

Background

Suppose we constrain a Turing machine to use no more space than what is initially occupied by the input. We can think of having a special endmarker symbol $\#$ so that the initial tape inscription has the form

$$\#x_1x_2\dots x_{n-1}x_n\#$$

The head is positioned at, say, x_1 and the transitions of the machine make it impossible to overwrite the endmarkers or to move past them. So, only n tape cells are available for the computation and a configuration consists of a word $w \in \Gamma^*$ plus a state and a head position. This type of machine is called a *linear bounded automaton (LBA)*. It is well-known that the acceptance problem for LBAs is PSPACE-complete.

In general, the tape alphabet is allowed to be larger than the input alphabet, so one can use standard coding tricks to free up an arbitrarily large fraction of the tape for computation, keeping a compressed version of the input in the other part of the tape.

Suppose we insist that the tape alphabet is exactly the same as the input alphabet, so this compression trick does not work any longer. Such a machine is said to be *overhead-free*.

Task

1. Describe an overhead-free automaton that recognizes palindromes over the binary alphabet.
2. Show that there is a PSPACE-complete language that is accepted by a overhead-free automaton.

Comment

2.8 Write-First Turing Machines

Background

It is customary to define Turing machines via a transition function of the form

$$\delta : Q \times \Gamma \rightarrow \Gamma \times \Delta \times Q$$

Here Q is the set of states, Γ the tape alphabet including a blank symbol, and $\Delta = \{-1, 0, +1\}$ indicates movement of the head. An instruction $\delta(p, a) = (b, d, q)$ indicates that the machine, when in state p and reading symbol a on the tape, will write symbol b , move the head by d and go into state q .

read — write — move — goto

Every action after the read depends on the symbol on the tape. This seems fairly natural, but there are other possibilities. For example, the machine could use a basic cycle

write — move — read — goto

The corresponding transition function has the format

$$\gamma : Q \rightarrow \Gamma \times \Delta \times (\Gamma \rightarrow Q)$$

Thus the machine writes a symbol and moves the head according to the current state. Only then will it read the tape (in a new position) and determine which state to move into. For the sake of clarity we refer to these machines as write-first Turing machines; their traditional counterparts will be called read-first Turing machines.

Task

1. Give a precise definition of what it means for a write-first Turing machine to compute a function.
2. Show that every write-first Turing machine can be simulated by a read-first machine.
3. Show that every read-first Turing machine can be simulated by a write-first machine.
4. How do the machines compare in size?

Comment

Assume for the sake of simplicity that the tape alphabet is $\Gamma = \{0, 1\}$.

2.9 Binary Register Machines

Background

A standard register machine operates on registers containing natural numbers. Apart from the logical operations there are only arithmetic operations such as increment and decrement that manipulate these numbers. Here is modified version of these machines: this time we operate directly on the bits in the binary expansion of the stored numbers. Thus we think of a register as a sequence of bits

$$R_r : b_0, b_1, \dots, b_n, \dots$$

Of course, only finitely many of these bits will be non-zero at any time and the numerical value of the register contents is $[R_r] = \sum_i b_i 2^i$.

A *binary register machine (BRM)* has the following instruction set:

- **zero r k l**
Check if $[R_r] == 0$; if so, goto instruction k , otherwise goto instruction l .
- **lshft r k**
Shift the contents of R_r to the left, goto instruction k .

- **rshft r k**
Shift the contents of R_r to the right (padding with 0), goto instruction k .
- **read r s k**
Read the first bit of R_r and write it into R_s , goto instruction k . Thus $[R_s] = 0$ or $[R_s] = 1$ depending on the first bit in R_r .
- **set r k**
Set the first bit of R_r to 1, goto instruction k .

Task

- Explain how to simulate a BRM on an ordinary RM. Make sure to give a separate argument for each of the BRM instructions.
- Explain how to implement a BRM instruction **unset r k** which sets the first bit of R_r to 0 (and continues at instruction k).
- Explain how to implement an increment operation **inc r k** on a BRM.
- Explain how to implement a conditional decrement operation **dec r k 1** on a BRM.
- Conclude that BRMs are computationally universal.
- Construct a universal BRM: a BRM that is capable of simulating all BRMs. You may safely assume that that the simulated machine has only one input register.

Comment

2.10 Reduced Turing Machines

Background

It is customary to define Turing machines via a transition function of the form

$$\delta : Q \times \Gamma \rightarrow \Gamma \times \Delta \times Q$$

Here Q is the set of states, Γ the tape alphabet including a blank symbol, and $\Delta = \{-1, 0, +1\}$ indicates movement of the head. An instruction $\delta(p, a) = (b, d, q)$ indicates that the machine, when in state p and reading symbol a on the tape, will write symbol b , move the head by d and go into state q .

Instead of using these fairly complex instructions we can simplify matters a bit by distinguishing several types of states.

- **Read:** for a read state p the machine scans the current tape symbol a and makes a transition into state $s(p, a)$.
- **Write:** for a write state p the machine writes $w(p)$ into the current tape cell and makes a transition into state p' .
- **Move:** for a left move state p the machine moves the head one cell to the left and makes a transition into state p' . Likewise for right move states.

So the state set in a reduced machine is partitioned into

$$Q = Q_R \cup Q_W \cup Q_I \cup Q_r$$

Task

1. Give a precise definition of what it means for a reduced Turing machine to compute a function.
2. Show that every ordinary Turing machine can be simulated by a reduced machine.
3. How do the machines compare in size?

Comment

This is just the tip of an iceberg. In the case where $\Gamma = \{0,1\}$ one can even insist that 1's are never overwritten by 0's (a so-called non-erasing TM), but the proof is rather complicated.

2.11 Semi-Decidable Sets and Computable Functions

Background

We defined semi-decidable sets in terms of semi-algorithms: on a Yes-instance the “algorithm” terminates but on a No-instance it keeps running forever.

There are many alternative definitions that exploit directly the relationship between semi-decidable sets and partial computable functions.

Task Below let $A \subseteq \mathbb{N}$. Show the following.

- A. A is semi-decidable iff it is the domain of a partial computable function.
- B. A is semi-decidable iff it is the range of a partial computable function.
- C. A is semi-decidable iff it is the range of an injective, partial computable function whose domain is an initial segment of \mathbb{N} .
- D. A is infinite and semi-decidable iff it is the range of an injective, partial computable function with domain \mathbb{N} .
- E. A is infinite and decidable iff there is a strictly increasing, partial computable function with range A and domain \mathbb{N} .

Comment

2.12 Graphs of Computable Functions

Background

Define the *graph* of a partial function $f : \mathbb{N} \rightarrow \mathbb{N}$ to be the set

$$\text{Gr}(f) = \{ (x, y) \mid f(x) \simeq y \} \subseteq \mathbb{N} \times \mathbb{N}$$

Task

- A. Show that a partial function is computable iff its graph is semi-decidable.
- B. Show that for any partial computable function f there is a partial computable function g such that for all x in the domain of f : $f(g(f(x))) = f(x)$.

Comment

2.13 Vertex Cover and SAT

Background

A classical problem in graph theory that is notoriously difficult computationally is to test whether a undirected graph has a Vertex Cover of size k . A vertex cover is any subset C of the vertex set such that every edge in the graph has at least one endpoint in C . There are lots of simple heuristic to construct a minimal size Vertex Cover, but none of them work properly in general. In fact, no algorithm is known that can check if G admits a Vertex Cover of size k in polynomial time (and there are good reasons to believe that in fact no such algorithm exists).

Task

- A. Given a graph $G = \langle V, E \rangle$ and a positive integer k , show how to translate the existence of a Vertex Cover of size k into a Satisfiability problem by constructing a Boolean formula Φ_G such that:

$$G \text{ has VC of size } k \iff \Phi_G \text{ is satisfiable.}$$

- B. Explain why your conversion algorithm that constructs the formula Φ_G from G in polynomial time (polynomial in $n + e$ where n is the number of nodes and e is the number of edges).

Comment

Make sure to explain what your Boolean variables mean and to prove that your construction really works as advertised. Note that this is completely different from trying to come up with a polynomial time algorithm to solve the Vertex Cover problem: you are just translating one hard problem into another, using limited computational resources in the translation.

2.14 Easy Satisfiability

Background

One can show that Satisfiability for propositional formulae is hard even if the given formula is in CNF and has exactly three literals per clause (3-CNF). No polynomial time algorithm is known that would solve the Satisfiability problem for formulae in 3-CNF (and most likely there is no such algorithm). However, three literals per clause is as far as one can go. A formula is said to be in 2-CNF if it has exactly two literals per clause:

$$\varphi = \{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_m, y_m\}\}$$

where the x_i and y_i are literals.

Task

- A. Show that one can test in polynomial time whether a formula in 2-CNF is satisfiable.
- B. Modify your algorithm so that it generates a satisfying truth assignment whenever such an assignment exists.
- C. Show that 2-CNF is NLOG-complete.

Comment

Use logic to rewrite the disjunctions as implications. Then think about chains of implications and use a graph algorithm.

Make sure to explain your algorithm precisely, analyze its running time and show that it is correct.

2.15 Partition

Background

The Partition Problem is an seemingly innocent decision problem

Problem: **Partition**

Instance: A list a_1, \dots, a_n of positive natural numbers.

Question: Is there a partition $[n] = A_1 \cup A_2$ such that $\sum_{i \in A_1} a_i = \sum_{i \in A_2} a_i$?

This is just the type of problem that yields to a dynamic programming approach (build some table row by row, then read off the result in the bottom right corner). Alas, the problem is known to be NP-complete, so all such attempts must ultimately fail (in the sense of not producing a really fast algorithm, but they may still produce reasonably good solutions).

Task

- A. Give a reasonable dynamic programming algorithm for Partition.

- B. Explain why your algorithm is correct and determine its running time.
- C. Given the fact that Partition is NP -complete, why does your algorithm not show that $\mathbb{P} = \text{NP}$?

Comment The last part assumes that you did not accidentally come up with the result of the century. My apologies if you did.

2.16 Kolmogorov Complexity

Background

We have seen in class that $K(x)$, the Kolmogorov complexity of x , is uncomputable (the proof being a version of Berry's paradox). As it turns out, even very weak approximations to K fail to be computable.

Task

1. Use Wurzelbrunft's alleged HaltingBlackBox to compute $K(x)$.
2. Show that there cannot be computable functions f and g such that

$$K(x) \leq g(x) \leq f(K(x)).$$

Comment Hint: For $f(z) = z$ this is just the uncomputability of K . Next, think about $f(z) = z + 100$. Generalize.

2.17 Uninspired Sets

Background

Let $C(x)$ be the Kolmogorov-Chaitin complexity of $x \in \mathbf{2}^*$. For any set $A \subseteq \mathbb{N}$ write $A_n = A \cap \{0, 1, \dots, n-1\}$ for the initial segment of A of length n . We can think of A_n as a binary string by applying some natural coding mechanism (for simplicity, let's say we use a bitvector of length n).

As we have seen, incompressibility with respect to Kolmogorov-Chaitin complexity is akin to randomness: there are no particular patterns one could exploit to obtain a shorter definition. How about the opposite notion? Call A *uninspired* if there is a constant c such that

$$C(A_n) \leq \log n + c.$$

So only some $\log n$ bits are needed to describe the corresponding bitvector of length n .

Task

- A. Show that any decidable set A is uninspired.
- B. How about the Halting Set K ? State whether K is uninspired and explain your reasoning.

Comment

The intuitive version of Kolmogorov-Chaitin is good enough for this application, you don't have to worry about prefix programs.

2.18 Expressiveness of FOL

Background

To deal with orders in FOL one can use a language L with a single binary relation symbol $<$. Recall that a total order is a well-order if there is no infinite descending chain

$$x_0 > x_1 > x_2 > \dots > x_n > \dots$$

Task

- A. Write down axioms in L that express: “the relation $<$ is a total order”.
- B. Show that the notion of a well-order is not axiomatizable in this setting: there is no set of axioms Γ in L such that

$$\langle A, < \rangle \models \Gamma \text{ implies } < \text{ is a well-order.}$$

Chapter 3

Induction, Iteration

3.1 Spreading Negativity

Background

Consider a cyclic sequence of five integers a_1, a_2, a_3, a_4, a_5 . We define an update operation on such sequences as follows: if a number is negative, add it to its two neighbors, then replace the number itself by its absolute value. For example, $1, 2, -5, 0, 1$ would turn into $1, -3, 5, -5, 1$. If there are multiple negative numbers we apply the rule in parallel, so the last sequence turns into $-2, 3, -3, 5, -4$. If there are no negative numbers the sequence does not change.

Let's refer to the sum $a_1 + a_2 + a_3 + a_4 + a_5$ as the *weight* $w(\vec{a})$ of the sequence.

Task

- A. Show that all sequences with positive weight will reach a fixed point.
- B. Prove or disprove the following conjecture: every starting sequence ultimately winds up in a limit cycle of length 1, 2 or 5.

Comment

Part (A) is truly annoying: once you see the key idea the rest can be handled by a computer algebra system, but without this idea there is little hope. I tend to believe that part (B) is true, but I am not certain (and, of course, I don't have a proof).

Lastly, note that length 5 is important here; this does not generalize to arbitrary lengths.

3.2 Hereditarily Finite Sets

Background

A *hereditarily finite set* is a set this is finite, contains only finite sets which in turn contain only finite sets, and so on. One can think of the collection of all hereditarily finite sets \mathbb{HF} as the universe of (finite)

combinatorics, given a few coding tricks such as Kuratowski's pairing function. Computations can also be modeled as operating over $\mathbb{H}\mathbb{F}$.

More formally, define $\mathbb{H}\mathbb{F}$ to be the smallest set H (with respect to set inclusion) that contains \emptyset and such that

$$x_1, \dots, x_n \text{ implies } \{x_1, \dots, x_n\} \in H. \quad (*)$$

Here is an alternative way of describing this collection of sets. By induction set $P_0 = \emptyset$ and $P_{n+1} = \mathfrak{P}(P_n) \cup P_n$ and let $\mathbb{P} = \bigcup P_n$.

$\mathbb{H}\mathbb{F}$ is countable and one can define a particularly simple bijection between \mathbb{N} and $\mathbb{H}\mathbb{F}$ as follows. By induction define $C : \mathbb{N} \rightarrow \mathbb{H}\mathbb{F}$ as $C(0) = \emptyset$ and $C(n) = \{C(i) \mid \text{dig}(n, i) = 1\}$ for $n > 0$. Here $\text{dig}(n, i)$ denotes the i th digit of n in the standard binary expansion, 0-indexed with the LSD in position 0.

Task

- A. Show that $\mathbb{P} = \mathbb{H}\mathbb{F}$.
- B. Show that C is a bijection.
- C. Give a reasonable description of C^{-1} .

Comment

3.3 Inverse Sequences

Background

One tool in the study of the sequence of primes $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \dots$ is the function counting the number of primes less than a given positive integer: $0, 1, 2, 2, 3, 3, 4, 4, 4, \dots$. Of course, the two sequences are very closely related: given the sequence of primes we can produce the counting sequence, and conversely.

More generally, suppose $f : \mathbb{N}^+ \rightarrow \mathbb{N}$ is non-decreasing and unbounded function. Define the associated counting function f^* by

$$f^*(x) = \text{number of } z : f(z) < x$$

Two functions f and g on the positive integers are complementary if, for all pairs m and n of positive integers

$$f(m) < n \text{ xor } g(n) < m.$$

Task

- A. Show that f^* is also non-decreasing and unbounded.
 - B. Show that the map $f \mapsto f^*$ is injective.
 - C. Show that f and f^* are complementary whenever f is non-decreasing and unbounded.
 - D. Determine f^{**} for a few examples such as the identity function, squares, Fibonacci numbers and the like. Form a conjecture as to what f^{**} is in general, and prove you conjecture.
-

3.4 Mystery Recursion

Background

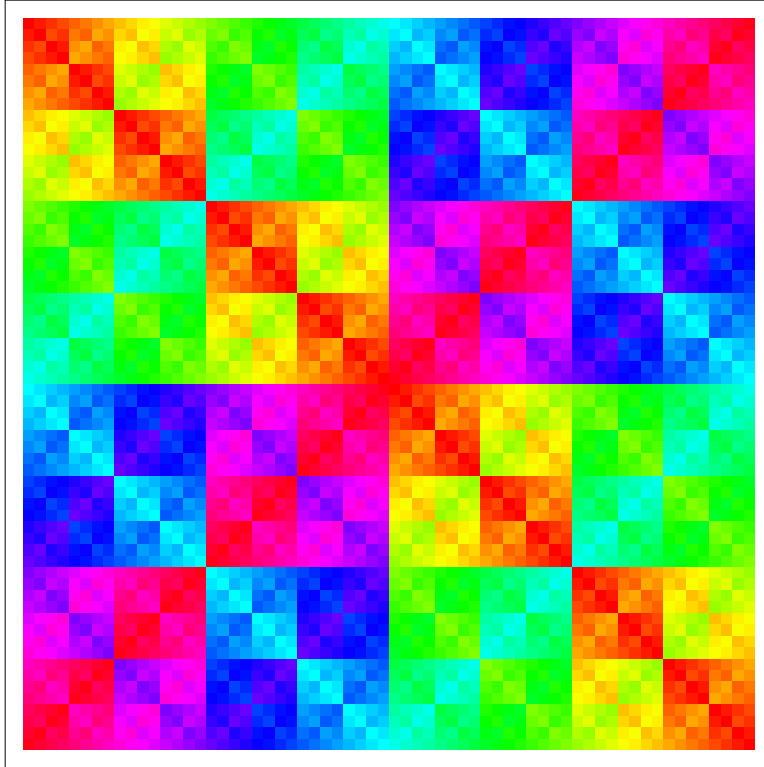
Here is a strange function $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Let $f(0, 0) = 0$ and set

$$f(x, y) = \min(z \mid z \neq f(x', y), x' < x \text{ and } z \neq f(x, y'), y' < y)$$

Here are the first few values of f .

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13
3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12
4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11
5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10
6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9
7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6
10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5
11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3
13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2
14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

There are obvious regularities which become much more visible in a color plot. The next picture covers 64×64 values.



Task

- A. Show that f is symmetric, that $f(x, 0) = x = f(0, x)$ and that $f(x, x) = 0$.
- B. Give a simple closed form for $f(x, y)$.

Comment

For the second part it may be a good idea to ignore the definition of f for a while and focus on the picture. Recall that it shows the values of $f(x, y)$ for $0 \leq x, y < 64$ and describe the recursive structure via a simple function of x and y . Then show that the new function is equivalent to f .

3.5 Nested Recursion

Background

Here is a function on the natural numbers, defined by a somewhat strange double recursion. Let $f(0) = 0$ and

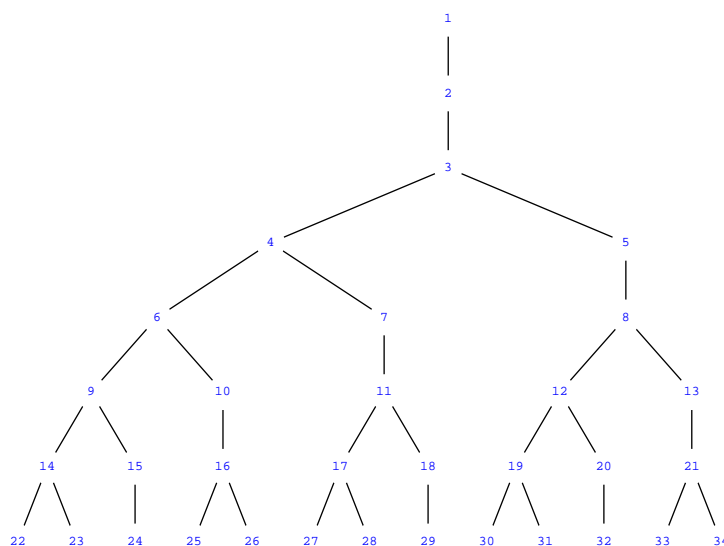
$$f(n) = n - f(f(n - 1)).$$

Task

- A. Show that f is defined for all natural numbers n .
- B. Show that f is monotonically increasing and that $f(n) - f(n - 1) \leq 1$.
- C. Show that the first 10000 values of the function can be written as $f(n) = \lfloor a \cdot (n + 1) \rfloor$ for some suitable constant a . Choose a wisely.
- D. Show that this closed form holds for all integers.
- E. **Extra Credit:** Use the Zeckendorf numeration system to give yet another simple description of $f(n)$.

Comment

Assuming you have found the right constant, to show that the floor representation is correct it helps to take a close look at the tree with edges $n \rightarrow f(n)$ below.



3.6 Powers of Two

Background Let $m < n$ be two positive integers. Recall that $\nu_2(x)$ denotes the maximum k such that 2^k divides x .

Claim: The interval $[m, n]$ contains a unique element that maximizes ν_2 .

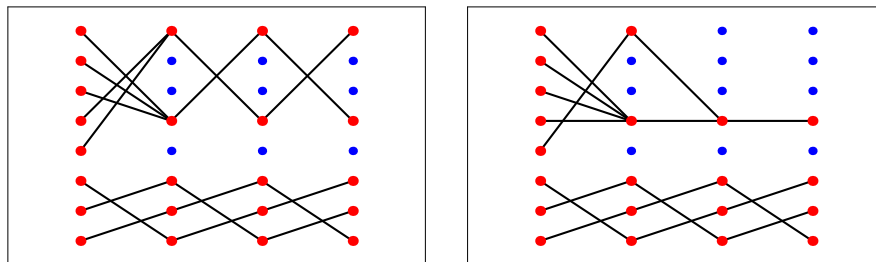
Task

- A. Prove the claim.
- B. Give an efficient algorithm that computes this unique element, given m and n .
- C. What is the time complexity of your algorithm?

3.7 Collapsing Transformations

Background

Consider a transformation $f : [n] \rightarrow [n]$ with kernel $K = (K_1, \dots, K_r)$ and range $I = (a_1, \dots, a_r)$. Here we assume that the kernel and image are ordered so that $f(K_i) = a_i$ and $a_1 < a_2 < \dots < a_r$. For an ordered kernel/image pair K, I let us write $\langle K, I \rangle$ for the corresponding transformation. The *collapse* of f , in symbols \hat{f} , is the map $[r] \rightarrow [r]$ given by $\hat{f}(i) = j \iff a_i \in K_j$. $\hat{f}(I)$ is the list $a_{\hat{f}(1)}, \dots, a_{\hat{f}(r)}$. We can think of \mathbb{T}_n , the collection of all transformations $[n] \rightarrow [n]$ as a monoid under composition. The kernel and image of a transformation is also helpful to test whether the transformation lies in a subgroup of \mathbb{T}_n .



As an example, consider $f = (2, 3, 1, 8, 8, 5, 5, 5) \in \mathbb{T}_8$, left in the picture. f has kernel $(\{3\}, \{1\}, \{2\}, \{6, 7, 8\}, \{4, 5\})$ and image $(1, 2, 3, 5, 8)$. The collapse is $\hat{f} = (2, 3, 1, 5, 4)$, a permutation of order 6. Indeed, f generates a subgroup of \mathbb{T}_8 of size 6. By contrast, $g = (2, 3, 1, 8, 5, 5, 5, 5)$, right in the picture, does not lie in a group, though g^2 does.

Task

- Show that for all $t \geq 1$: $f^t = \langle K, \hat{f}^{t-1}(I) \rangle$.
- A transformation f lies in a subgroup of \mathbb{T}_n iff its image is a selector of its kernel (meaning that $|K_i \cap I| = 1$ for all i). Moreover, the cardinality of the subgroup G generated by f is the order of \hat{f} in the symmetric group on r points.

Comment

3.8 Reversal and Palindromes

Background

Define the reversal of a word over alphabet Σ inductively as follows.

$$\begin{aligned} \text{rev}(\varepsilon) &= \varepsilon \\ \text{rev}(xa) &= a \text{rev}(x) \end{aligned}$$

Here $x \in \Sigma^*$, $a \in \Sigma$. A word x is a palindrome if $x = \text{rev}(x)$.

Task

- A. $\text{rev}(ax) = \text{rev}(x) a$ for all $x \in \Sigma^*$, $a \in \Sigma$.
- B. $\text{rev}(xy) = \text{rev}(y) \text{rev}(x)$ for all $x, y \in \Sigma^*$.
- C. All palindromes are of the form $w \text{rev}(w)$ or $wa \text{rev}(w)$.
- D. $\text{rev}(\text{rev}(x)) = x$ for all $x \in \Sigma^*$.
- E. $\text{rev}(x^n) = \text{rev}(x)^n$ for all $x \in \Sigma^*$, $n \geq 0$.

Comment

3.9 Lookup Tables and Iteration

Background

Suppose a function $f : [n] \rightarrow [n]$ is given as a lookup table. Assume that n is machine-sized so that $f(x)$ can be determined in $O(1)$ time. In order to determine $f^t(x)$ we can use repeated lookup. However, if we need to determine $f^t(x)$ repeatedly it is better to perform a pre-computation that augments the table for f . The augmented table then allows for speedy lookups of iterated values of the function f .

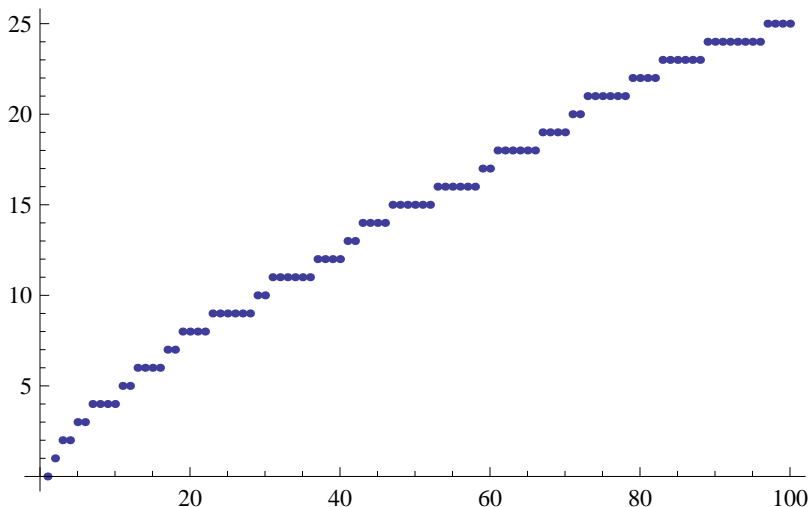
Task

- A. Explain how to augment the table for f so that lookups of $f^t(x)$ are fast.
 - B. How long does it take in your solution to determine $f^t(x)$?
 - C. What is the memory requirement for your table?
 - D. What is the cost of the pre-computation?
-

3.10 Iteration and Composites

Background

The number theoretic function $\pi(n)$ is defined to be the number of primes up to and including n . Asymptotically, the function grows like $n/\ln n$ but close-up its behavior is quite erratic.



For any $n \geq 1$ define $f_n(x) = \pi(x) + n$ and let $F(n)$ be the fixed point of n under f_n . Somewhat surprisingly, the first few values of F are 1, 4, 6, 8, 9, 10, 12, \dots , all the composites (in the sense of “not prime,” whence 1 is included).

Task

- A. Show that the fixed point used to define $F(n)$ always exists.
- B. Show that F enumerates the composite positive integers.
- C. Explain why this method works.

Comment

Note that it is quite difficult to compute $\pi(n)$ exactly for large n . Approximations are easy to get.

3.11 Fast Exponentiation

Background

Several classical fast methods to perform fast exponentiation a^e where e is a positive integer (and a an element of some monoid) are based on squaring:

$$a^{2e} = \text{sq}(a^e) \quad a^{2e+1} = \text{sq}(a^e) \cdot a$$

Here are three variants.

// exp1

```

x = 1;
foreach d binary digit of e, MSD first do
    x = sq(x);
    if( d == 1 ) x = a * y;
return x;

```

```

// exp2
(x,z) = (1,a);
foreach d binary digit of e, LSD first do
    if( d == 1 ) x = x * z;
    z = sq(z);
return x;

// exp3
(x,z) = (1,a);
foreach d binary digit of e, MSD first do
    if( d == 1 )
        (x,z) = ( x * z, sq(z) );
    else
        (x,z) = ( sq(x), x * z );
return x;

```

Task

- Show that `exp1` correctly computes a^e .
 - Show that `exp2` correctly computes a^e .
 - Show that `exp3` correctly computes a^e .
 - Compare the performance of the three methods. Distinguish between multiplications and squaring operations.
-

3.12 Greatest Common Divisor

Background

The classical method to compute greatest common divisors of two positive integers is the Euclidean algorithm. Here are two alternative ways to compute the GCD. The first is slightly non-standard but the second is positively wild. Assume that n and m are positive integers.

Here is the first function:

$$H(n, m) = \begin{cases} m & \text{if } n = 0 \\ 2H(n/2, m/2) & \text{if } n, m \text{ even} \\ H(n/2, m) & \text{if } n \text{ even, } m \text{ odd} \\ H(n, m/2) & \text{if } n \text{ odd, } m \text{ even} \\ H((n-m)/2, m) & \text{if } n, m \text{ odd, } n \geq m \\ H(n, (m-n)/2) & \text{if } n, m \text{ odd, } n < m \end{cases}$$

And here is the second:

$$G(n, m) = 2 \sum_{i=1}^{m-1} \lfloor \frac{ni}{m} \rfloor + n + m - nm$$

Task

- A. Show that $H(n, m)$ is the GCD of n and m .
- B. Show that $G(n, m)$ is the GCD of n and m .
- C. Determine the time complexity of both functions.

Comment

Note that it is absolutely unclear from the definition that G is symmetric. It might help to implement the function and compute the terms in the summation in a few concrete cases.

3.13 Binary Square Roots

Background

For positive integers n define $\beta(n) = \text{binsqr}(n, 1)$ where the latter function is defined by

```
binsqr( n, s ) {  
    if( n < s ) return 0;  
    z = binsqr(n, 2*s) + s;  
    return ( z*z <= n ? z : z - s );  
}
```

Task

- A. Show that $\beta(n) = \lfloor \sqrt{n} \rfloor$.
 - B. What is the time complexity of β (assuming the standard multiplication method).
-

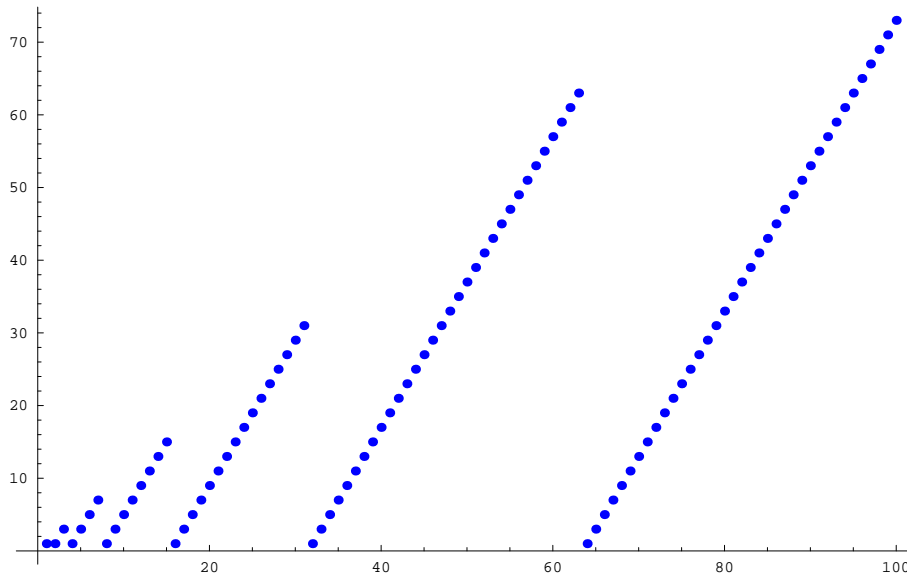
3.14 The Josephus Problem

Background

We forgo the opportunity to present a bloody cover story. Here is the problem in terms of list operations. Define a decimation (well, 10 is 2 in binary after all) operation Δ on non-empty lists as follows:

$$\Delta(x_1, \dots, x_n) = (x_3, x_4, \dots, x_n, x_1)$$

Thus, after a cyclic shift to the left the first element is dropped. For completeness, let $\Delta(x) = ()$. Also, for any list L and $t \geq 1$ let $J(t, L)$ be the element omitted from $\Delta^t(L)$ and let $J(n) = J(n, (1, \dots, n))$. The first 100 values of $J(n)$ can be seen in the plot below.



Task

- A. Explain why $J(2^k) = 1$.
- B. Find an elegant description of $J(n)$ for general n . Take a close look at the plot above to get started and think about binary expansions.

Comment

There are natural generalizations that you might want to think about. Things become quite messy, though.

3.15 Thue and Shuffle

Background

Recall the definition of the Thue sequence.

$$\begin{aligned}
 t_0 &= 0 \\
 t_{2n} &= t_n \\
 t_{2n+1} &= \overline{t_{2n}}
 \end{aligned}$$

Let's write T_n for the binary word consisting of the first n bits of the Thue sequence. For example,

$$T_{32} = 0110100110010110100101100110100110010110011010010110100110010110$$

There are many alternative ways to describe the Thue sequence; here are two of them. First, recall the shuffle operation which interleaves the elements of two sequences of equal length in a strictly alternating fashion:

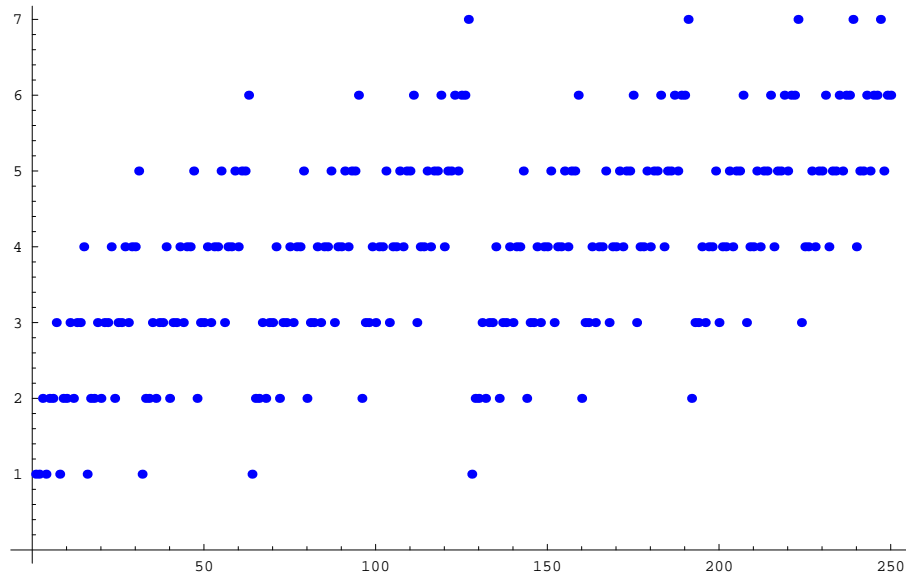
$$\text{sh}(\vec{a}, \vec{b}) = a_1b_1a_2b_2 \dots a_{k-1}b_{k-1}a_k b_k$$

Given σ we can define a sequence of binary words as follows.

$$S_0 = 0$$

$$S_n = \text{sh}(S_{n-1}, \overline{S_{n-1}})$$

The *binary digit sum* $\sigma(n)$ of a non-negative integer n is the sum of the binary digits of n . In other words, $\sigma(n)$ is the number of 1's in the binary expansion of n . σ jumps around quite a bit, here are the first 250 values.



Task

- A. Show that $t_n = \sigma(n) \pmod{2}$.
- B. Show that $T_{2^k} = S_k$.

3.16 Fibonacci Words

Background

The following recursion defines a collection of binary words, the so-called Fibonacci words.

$$W_0 = \varepsilon$$

$$W_1 = 0$$

$$W_2 = 1$$

$$W_n = W_{n-1} W_{n-2}$$

The first 10 Fibonacci words are listed in the following table:

0	ε
1	0
2	1
3	10
4	101
5	10110
6	10110101
7	1011010110110
8	10110101101101010101
9	101101011011010110101101011010110110

Needless to say, the lengths of the Fibonacci lists are just the Fibonacci numbers. Note that $W_{2n} = \dots 01$ and $W_{2n+1} = \dots 10$ for $n \geq 2$. For any word of length at least 2 define $\sigma(x_1x_2 \dots x_{k-1}x_k)$ to be the word $x_1x_2 \dots x_{k-2}x_kx_{k-1}$. Let $\delta(x_1x_2 \dots x_{k-1}x_k) = x_1x_2 \dots x_{k-2}$. Thus operation σ swaps the last 2 letters of a word while δ removes the last 2 letters.

Task

- A. Show that $W_n = \sigma(W_{n-2}W_{n-1})$ for all $n \geq 3$.
- B. Show that $\delta(W_n)$ is a palindrome for all $n \geq 4$.

3.17 Tournaments and Kings

Background

A tournament is a random orientation on a complete graph: We interpret edge $x \rightarrow y$ to mean that x beats y . Call a node v a *king* if for any node x there is a path from v to x of length at most 2. A royal tournament is one where everybody is a king. A *champion* is any node with maximal out-degree: a player with the maximal number of wins. Hence champions trivially exist, though they are not unique in general. It is not clear that kings exist in general.

Claim: Every tournament has a king.

Task

- A. Prove the claim by induction on the number of nodes.
- B. Prove the claim by showing that every champion is a king.
- C. Show that a royal tournament of size n exists if, and only if, $n \neq 2, 4$.

3.18 Tournaments and Fairness

Background

A tournament is a random orientation on a complete graph: We interpret edge $x \rightarrow y$ to mean that x beats y . Suppose the tournament has vertex set V and size n . By a *ranking* we mean a function $r : V \rightarrow [n]$. Intuitively, a ranking is fair if whenever there is a path from p to q then p is higher ranked: $r(p) < r(q)$. Alas, tournaments may well contain cycles: “beats” is not a transitive relation and so in general we cannot construct fair rankings. Call a ranking 1-fair if $r(p) = r(q) + 1$ implies that q beat p .

Task

- Show how to construct a fair ranking in linear time for any acyclic tournament.
- Show that any tournament that contains a cycle of length at least 3 also contains a cycle of length exactly 3.
- Show that every tournament admits a 1-fair ranking.

Comment

The second part is tricky; use induction on the number of nodes to show the stronger claim that the top-ranked node can always be chosen to be a champion (maximum number of wins).

3.19 Ducci Sequences

Background

Recall the Ducci sequences introduced in class: iterate the function

$$D(x_1, x_2, x_3, x_4) = (|x_1 - x_2|, |x_2 - x_3|, |x_3 - x_4|, |x_4 - x_1|)$$

on \mathbb{N}^4 . A typical orbit looks like so:

0		94	68	11	85
1		26	57	74	9
2		31	17	65	17
3		14	48	48	14
4		34	0	34	0
5		34	34	34	34
6		0	0	0	0

Task

- Explain how to use the tribonacci numbers

$$t_n = t_{n-1} + t_{n-2} + t_{n-3} \quad t_0 = t_1 = 0 \quad t_2 = 1$$

to show that the transients (leading to fixed point $\mathbf{0}$) can be arbitrarily long.

- B. Prove that every Ducci sequence ends in fixed point $\mathbf{0}$. We are only dealing with vectors of length 4 here.

Comment

You probably want to write a little program and print out some orbits of tribonacci Ducci-orbits; ponder deeply and a conjecture will jump out at you. Even with the right conjecture the proof is a bit tedious. Try to come up with a crisp, elegant argument.

Extra credit: Generalize the second claim to any initial vector of length 2^k .

3.20 The DAZS Operator

Background

For this problem, consider non-decreasing lists of positive integers $A = (a_1, a_2, \dots, a_w)$. We transform any such list into a new one according to the following simple recipe:

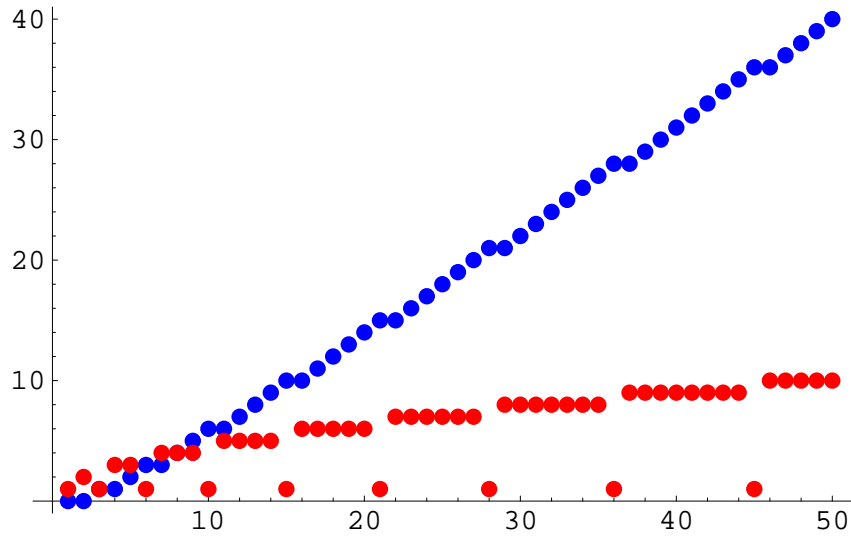
- Subtract 1 from all elements.
- Append the length of the list as a new element.
- Sort the list.
- Remove all 0 entries.

We will call this the DASZ operation (decrement, append, sort, kill zero) and write $D(A)$ for the new list (note that D really is a function). For example, $D((1, 3, 5)) = (2, 3, 4)$, $D((4)) = (1, 3)$ and $D((1, 1, 1, 1)) = (4)$.

A single application of D is not too fascinating, but things become interesting when we apply the operation over and over again. As it turns out, no matter what the starting point A is, we always have $D^{t+p}(A) = D^t(A)$ for some $t \geq 0$ and $p > 0$. The least t and p are called the transient and period of A , respectively. For example, the transient and period of $(1, 1, 1, 1, 1)$ are both 3:

0	1	1	1	1	1	1
1	5					
2	1	4				
3	2	3				
4	1	2	2			
5	1	1	3			
6	2	3				

Recall that a fixed point of D is any list A such that $D(A) = A$ (i.e. the transient is 0 and the period is 1). The picture below shows the transients and the periods of all starting lists $A = (n)$ for $n \leq 50$. In the picture, blue indicates the transients and red the periods. The lists leading to a fixed point (corresponding to period 1) produce the red dots at the bottom of the picture.



Task

- A. Explain why lists must repeat after a finite number of steps (i.e., it cannot happen that $D^i(A) \neq D^j(A)$ for all $i < j$).
- B. Characterize all the fixed points of the DASZ operation.
- C. Determine which initial lists $A = (n)$ lead to a fixed point.

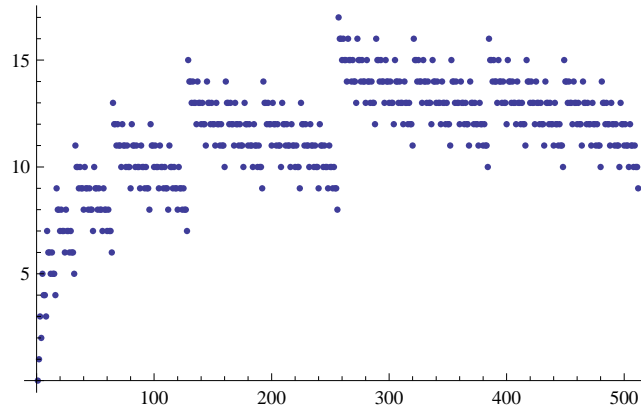
3.21 UnCollatz

Background

Here is a function on the positive integers that looks quite similar to the infamous Collatz C function.

$$U(x) = \begin{cases} 1 & \text{if } x = 1, \\ x/2 & \text{if } x \text{ even,} \\ x + 1 & \text{otherwise.} \end{cases}$$

Unlike with the Collatz function, though, it is easy to show that all orbits of U lead to the fixed point 1. Hence we can define the stopping time $\sigma(x)$ of x to be the number of steps U takes to reach the fixed point. A plot of the first 512 values is below. Note the nice fractal structure.



Task

- Show that any orbit of U ends in the fixed point 1.
- Give as simple a description of the stopping time as you can manage.
- Describe the distribution of stopping times for all k bit numbers.

Comment

For the second and third part some experimentation is probably helpful; it's a bit tricky to get a completely correct answer.

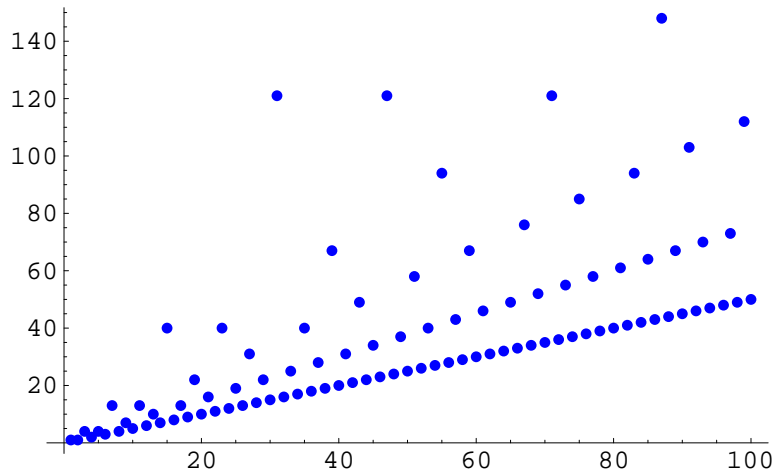
3.22 Son of Collatz

Background

A function defined by an apparently simple recursion can behave in a rather unpredictable way. The Collatz function and the Ackermann function are prime examples. Here is an example that looks somewhat similar. Define a function F on the positive integers by

$$F(x) = \begin{cases} x/2 & \text{if } x \text{ is even,} \\ F(F(3x + 1)) & \text{otherwise.} \end{cases}$$

Note the double application of F in the odd case. It is not really clear that this is well-defined, there might be some infinite loop – but there isn't. Here is a plot of F on up to $x = 100$.



Task

- Determine what the lines in the picture are. More precisely, determine a simple description of the x-coordinates of all the points belonging to a single one of these lines (the y-coordinates are then easy to get).
- Give a reasonably simple non-recursive description of F .
- Prove that your description is correct, and conclude that F is really well-defined: for any positive integer x there is exactly one y such that $F(x) = y$.
- Define $d(x)$ to be the number of recursive calls made in the computation of $F(x)$. For example, for all even x , $d(x) = 0$, $d(1) = 2$ and $d(3) = 4$. Find a simple description of d .

Comment

It is a good idea to try to figure out how this function is related to the Collatz function C . A little experimental computation might also be helpful.

3.23 Floyd's Cycle Detection Trick

Background

Recall Floyd's cycle finding algorithm that allows one to compute the period of a point under some function $f : A \rightarrow A$ where A is a finite ground set. In the classical version, the two pebbles move at speed $u = 1$ and $v = 2$, respectively, until they meet on the cyclic part of the orbit.

Task

- Consider the choice $u = 2$, $v = 3$ for the speed of the pebbles. Does the algorithm still work for all possible inputs? Justify your answer.
- Try to generalize to other values of u and v : when does the algorithm work, when does it fail?

- C. For those values of u and v where the algorithm works, how does the running time compare to the standard version $u = 1$ and $v = 2$? By running time we mean the number of applications of f .
-

3.24 Floyd and Teleportation

Background

Recall Floyd's trick to determine the transient and period of a finite orbit without using extra memory: two pebbles move at speeds 1 and 2, respectively. Here is a modification of this approach where the "slow" pebble does not move at all, except that it occasionally teleports to the location of the "fast" pebble.

Here is the crucial first part of the algorithm that computes the period of the orbit. The function in question is f and the starting point is a .

```
slow = a;
fast = f(a);
per = bnd = 1;
while( fast != slow )
    if( per == bnd )
        { slow = fast; per = 0; bnd *= 2; }
    fast = f(fast);
    per++;

return per;
```

The claim is that this algorithm finds a point on the limit cycle. In a separate procedure one can then compute the transient (just as in Floyd's method).

Task

- Prove that upon termination the pebbles are located on some position on the limit cycle. Conclude that the return value is indeed the period of the orbit.
- Determine the time complexity of the teleportation algorithm (i.e., count the number of times the loop is executed).
- Now consider the task of computing both transient and period. Compare the performance of Floyd's classical algorithm and the teleportation algorithm. Is one better than the other? Why?

Comment

This method is due to R. Brent.

3.25 Schröder-Bernstein

Background

Suppose $f : A \rightarrow B$ and $g : B \rightarrow A$ are two injective functions. Define

$$\begin{aligned}A_0 &= AA_{n+1} = g(B_n) \\ B_0 &= BB_{n+1} = f(A_n)\end{aligned}$$

and let $A_\omega = \bigcap A_n$.

Task

- A. Explain the sets $A_{2n} - A_{2n+1}$ and A_ω .
- B. Define a new function $h : A \rightarrow B$ by

$$h(x) = \begin{cases} f(x) & \text{if } x \in A_\omega \text{ or } x \in A_{2n} - A_{2n+1}, \\ g^{-1}(x) & \text{otherwise.} \end{cases}$$

Show that h is a bijection.

Comment

3.26 Cardinalities

Background

For the following problem you should rely on the Schröder-Bernstein theorem rather than trying to construct bijections directly.

Task

- A. Show that the real intervals $[0, 3]$, $[0, 2)$ and $(0, 1)$ all have the same cardinality.
- B. Show that $|2^\omega| = |\omega^\omega|$.

Comment

Chapter 4

Combinatorics

4.1 Flattening Multisets

Background

A *finite multiset* over a set A can be thought of as a map $\alpha : A \rightarrow \mathbb{N}$ such that $\alpha(a) > 0$ for only finitely many $a \in A$. Intuitively, $\alpha(a)$ is the multiplicity of object a in α . The *weight* of α is $\sum \alpha(a)$. Call a multiset *flat* if it represents a plain set: $\alpha(a) \leq 1$ for all a .

We are here interested in finite multisets over the natural numbers. Let's consider two operations on these multisets, *shift* and *cancel*. The shift operation transforms α into β provided that for some k , $\alpha(k) \geq 2$ and β agrees with α except that $\beta(k) = \alpha(k) - 2$, $\beta(k+2) = \alpha(k+2) + 1$, $\beta(k+3) = \alpha(k+3) + 1$. For cancel we have for some k , $\alpha(k) \geq 2$, $\alpha(k+1) \geq 2$ and $\alpha(k+2) \geq 1$; β agrees with α except that $\beta(k) = \alpha(k) - 2$, $\beta(k+1) = \alpha(k+1) - 2$ and $\beta(k+2) = \alpha(k+2) - 1$.

We say that α reduces to β if there is a sequence of shift and cancel operations that leads from α to β . For example,

$$\begin{aligned} (3, 2, 0, 0, 0, 0, 0, \dots) &\xrightarrow{\text{shift}} \\ (1, 2, 1, 1, 0, 0, 0, \dots) &\xrightarrow{\text{shift}} \\ (1, 0, 1, 2, 1, 0, 0, \dots) &\xrightarrow{\text{shift}} \\ (1, 0, 1, 0, 1, 1, 1, 0, \dots) & \end{aligned}$$

and

$$\begin{aligned} (3, 0, 1, 1, 1, 0, 0, \dots) &\xrightarrow{\text{shift}} \\ (1, 0, 2, 2, 1, 0, 0, \dots) &\xrightarrow{\text{shift}} \\ (1, 0, 0, 2, 2, 1, 0, \dots) &\xrightarrow{\text{cancel}} \\ (1, 0, 0, 0, 0, 0, 0, \dots) & \end{aligned}$$

We will establish the following

Claim 1: Every finite multiset of natural numbers reduces to a flat multiset under the operations of shift and cancel.

The following function will be useful in the proof of Claim 1. Let $f : \mathbb{N}^3 \rightarrow \mathbb{N}^3$ be defined by $f(x, y, z) = (y, z + x/2, x/2)$ where $x/2$ denotes integer division.

Claim 2: The map f is 2-to-1 and all orbits of f end in fixed points of the form $(2i, 2i, i)$.

For example, the following orbit of f ends in $(2, 2, 1)$.

$$(2, 3, 4), (3, 5, 1), (5, 2, 1), (2, 3, 2), (3, 3, 1), (3, 2, 1), (2, 2, 1)$$

Task

- A. Show that the shift operations alone is not enough to reduce an arbitrary finite multiset to a flat multiset.
- B. Show that Claim 1 follows from Claim 2.
- C. Write $f^n(x, y, z)$ as three rational polynomials in x, y, z and find a recurrence equation for the coefficients of these polynomials. Hint: compute a few values and find a pattern.
- D. Using this characterization of the polynomials prove Claim 2. Hint: assume $f^n(x, y, z) = (x, y, z)$ for some $n > 0$.

Comment Appearances notwithstanding, this problem is not artificial but arises naturally in a proof that shows that a certain monoid of functions is isomorphic to the commutative free group on two generators.

4.2 Pruning Labels

Background

In an ordered tree T any two nodes x and y must have a uniquely determined *lowest common ancestor (LCA)*: the lowest node in T that has both nodes as descendants. Thus, the LCA is the node z such that the subtree $T(z)$ contains both x and y but this property fails for all the children z' of z . A node x is said to be *to the left of* y if they do not lie on the same branch and, letting z be the LCA with children z_1, \dots, z_k we have $x \in T(z_i)$ and $y \in T(z_j)$ for some $i < j$.

Here is a problem that occurs in an important algorithm in automata theory. Consider finite ordered trees whose nodes are labeled by subsets of some finite set Q ; we write $\lambda(x)$ for the label of x . We need to remove all elements from $\lambda(y)$ that also occur in some label $\lambda(x)$ where x is to the left of y . We will call this process *horizontal pruning*.

Also, we would like to remove the labels of all descendants of y and refer to this as *vertical pruning*.

Task

- A. Find an algorithm to do horizontal pruning that runs in time linear in the number of tree nodes, assuming that the set operations are constant time.
- B. Find an algorithm to do vertical pruning that runs in time linear in the number of tree nodes, assuming that the set operations are constant time.

- C. Find a way to combine your two algorithms into one (of course, that does not mean to compose them sequentially).

Comment

Make sure to prove that your algorithms are correct.

4.3 Power Sums

Background

Let us write

$$P(n, k) = \sum_{i=1}^n i^k$$

for power sums. It is not hard to see that $P(n, k)$ is a polynomial of degree $k + 1$ in n , but finding the coefficients is a bit more problematic. Here is a elegant but somewhat mysterious approach. Let's suppose that

$$(S - 1)^n = S^n$$

Don't worry, keep reading. We now can write

$$(k + 1)i^k = (i + S)^{k+1} - (i - (S - 1))^{k+1}$$

Rewrite $P(n, k)$ as a telescoping sum and get

$$P(n, k) = \frac{(n + S)^{k+1} - S^{k+1}}{k + 1}$$

A nice closed form except for the little problem with S .

Task

- A. Show that $P(n, k)$ is a polynomial of degree $k + 1$ in n .
- B. Explain how to interpret S^i as an unknown S_i and exploit this interpretation to compute the polynomial for $P(n, 10)$.

Comment

This involves solving a few linear equations, you probably do not want to do this by hand.

4.4 3,5,7-Ordered Sequences

Background

Let A be any sequence of distinct integers of length n . A is h -ordered if $A_i < A_{i+h}$ for all $i \leq n - h$. So any 1-ordered sequence is also sorted. There is a famous sorting algorithm due to D. L. Shell in 1959 that sort as list by performing repeated insertion sorts on sublist of different strides. For example, a variant suggested by Knuth would sort a list of length 10000 by making it h -sorted for $h = 9841, 3280, 1093, 364, 121, 40, 13, 3, 1$. Incidentally, the analysis of Shell-sort is excruciatingly difficult.

This is the type of question that can be solved by some informed trial-and-error. What we are after here, though, is a more systematic attack. To this end, we consider the local order structure of a sequence. Consider the blocks of width 5 of the sequence, and replace them by their order type. E.g., the sequence

50, 52, 56, 75, 55, 18, 77, 15

produces

(1, 2, 4, 5, 3), (2, 4, 5, 3, 1), (3, 4, 2, 1, 5), (4, 3, 2, 5, 1)

Note that there are restrictions on how these blocks can follow one another.

Task

- A. Analyze the local order structure of 3,5-ordered sequences.
- B. Use your results to construct a 3,5,7-ordered sequence of length 10^6 that has more than 10^6 inversions.

Comment

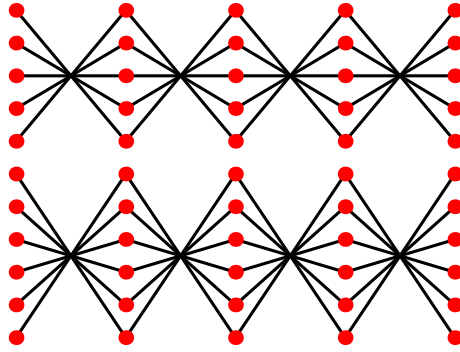
4.5 A Permutation

Background

Here is a definition of a class of permutations in terms of a small program. More precisely, we define functions $f_k : (n) \rightarrow (n)$ where, as always, $(n) = \{0, 1, \dots, n - 1\}$. Pseudo-code for f_k :

```
f( k, x ) {  
    z = n - x - 1 - k;  
    return ( z >= 0 ? z : z + n );  
}
```

The intended range for parameters k and x is (n) . Here is a plot of one such function, for $n = 11$ and $k = 5$.



It is clear from the picture that f_5 is a permutation, has one fixed point and all other points lie on 2-cycles.

Task

- A. Show that f_k is a permutation for all n and $0 \leq k < n$.
- B. Assume n odd. Show that f_k has exactly one fixed point and all other points lie on 2-cycles.
- C. What happens for even n ?
- D. Fix n . The permutations f_k naturally act on binary lists of length n . How many fixed points are there?

4.6 Kraft's Inequality

Background

Kraft's inequality is a combinatorial lemma that is often used in the theory of prefix codes; it is also crucial for the construction of Chaitin's Ω .

Kraft's Lemma:

Let $S \subseteq \mathbf{2}^*$ be a prefix set of binary words. Then

$$\sum_{x \in S} 2^{-|x|} \leq 1$$

On the other hand, given natural numbers ℓ_n such that

$$\sum_{n \geq 1} 2^{-\ell_n} \leq 1$$

there exists a prefix set $S = \{s_n \mid n\} \subseteq \mathbf{2}^*$ such that $\ell_n = |s_n|$. S is said to realize (ℓ_n) . For example, $\ell_n = n$ produces $\sum 2^{-\ell_n} = 1$ and can be realized by $s_n = 0^{n-1}1$.

Task

- A. Prove the first part for any finite set S .

- B. Prove the second part for any finite set S .
- C. Conclude that the lemma holds for arbitrary sets.

Comment

Try to be as algorithmic in the second part as possible. For the sake of TA sanity, let's all assume that (ℓ_n) is an ordered sequence.

What exactly is needed to make the transition to the infinite case?

4.7 Non-Decreasing Functions

Task

In class we showed how to rank and unrank injective functions as well as strictly increasing functions $f : [m] \rightarrow [n]$. This problem deals with slightly different class: non-decreasing functions (so $f(i) \leq f(i+1)$).

Task

- A. How many non-decreasing functions are there in terms of m and n ?
- B. Describe a good ranking and unranking function for the class of non-decreasing functions.
- C. What are the time complexities of your rank and unrank operations?

Comment

Try to come up with an elegant solution, it should be easy to describe what you are doing. If you are using a complicated method make sure to prove that it is correct.

4.8 Tic-Tac-Toe

Task

Use the material in class to give a complete solution for the odious Tic-Tac-Toe problem. Let $G \subseteq S_9$ be the required group acting on the board.

1. Determine the cycle structure of all the elements of G .
 2. Determine the number of patterns with 3 naughts and 3 crosses.
 3. Determine the complete pattern inventory.
-

4.9 Generating Permutations

Background

Here is one of the standard algorithms for a successor operation on permutations of $[n]$: given as input a permutation $A = (a_1, a_2, \dots, a_n)$ of $[n]$, the algorithm will compute the “next” permutation. Here is an informal description:

- Find the largest position $i < n$ such that $a_i < a_{i+1}$.
- Find the least element $a_j > a_i$ in positions $i < j \leq n$.
- Swap a_i and a_j .
- Sort the tail-end a_{i+1}, \dots, a_n of the permutation.

This works as long as the position i in the first step exists. Otherwise the whole permutation is already descending, i.e., $A = (n, n-1, \dots, 2, 1)$, in which case we wrap around and return the identity permutation. Below are the first few iterations of the algorithm on the identity permutation for $n = 4$.

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
```

You might want to implement this algorithm so you are absolutely clear how it works.

Task

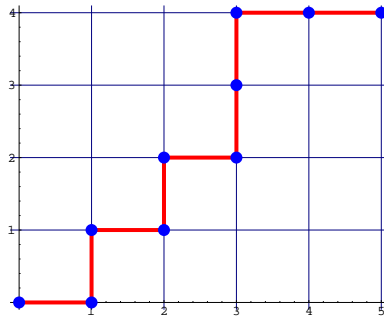
- Show that this algorithm is correct: the orbit of any permutation on $[n]$ under this operation consists of all permutations of $[n]$.
- Carefully explain the running time of this operation. Any clever ideas?

4.10 Manhattan Paths

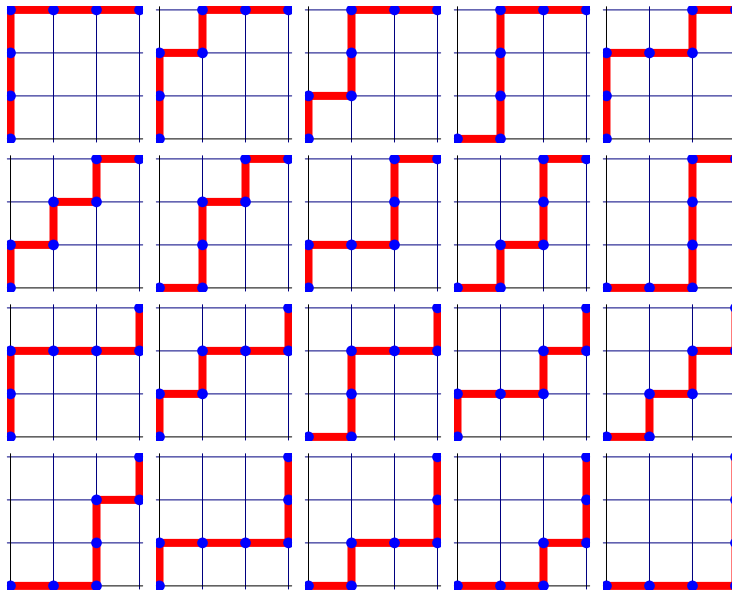
Background

A standard counting trick for problems involving binomial coefficients is to use the fact that the number of North-East-paths from $(0, 0)$ to (n, m) in an n by m grid is $\binom{n+m}{n}$. By counting these path in clever ways one can often establish identities involving binomial coefficients that are otherwise difficult to establish.

Here is a picture of one specific path on a 4 by 5 grid.



And a picture of all 20 paths in a 3 by 3 grid.



Note that the paths are generated by repeated application of a “next” operation. Essentially this is just the plot of the orbit of this function.

Task

- A. Use path counting to establish the identity

$$\sum \binom{n}{i}^2 = \binom{2n}{n}$$

- B. Explain how the next operation used to generate the last picture works.

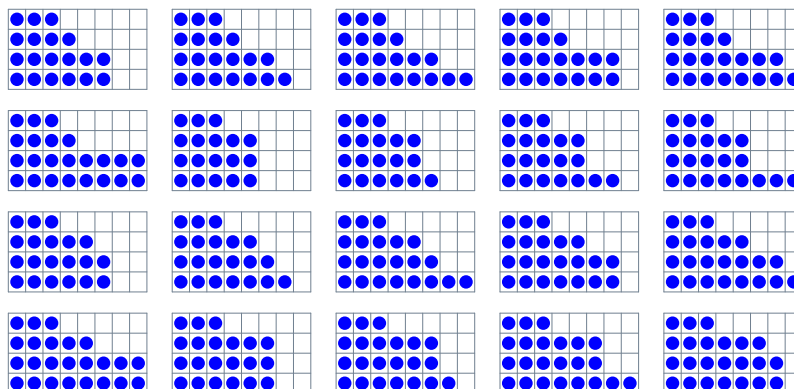
4.11 Chomp

Task

Chomp is an impartial 2-person game that has very nice theoretical properties – but is rather boring in actual play. In the 2-dimensional version, you start with an n by m chocolate bar, subdivided as usual in nm smaller rectangles. We give natural coordinates to these smaller pieces, (x, y) where $1 \leq x \leq n$ and $1 \leq y \leq m$.

A move consists of the following: the player chooses a small piece (x, y) that has not already been removed. She then bites off this piece, as well as all pieces (x', y') where $x \leq x' \leq n$ and $y \leq y' \leq m$. The players alternate taking moves. The problem is that piece $(1, 1)$ is poisoned, so whoever is forced to take that last piece loses.

The picture shows some possible configurations during a game played on a 8 by 4 chocolate bar.



Make sure you understand what possible sequence of moves could have produced these configurations. Also, think about which configurations cannot arise (some U-shaped arrangement, for example).

Task

- How many non-decreasing functions from $[m]$ to $[n]$ are there? The answer should be a simple binomial expression.
- How many possible configurations are there for a game played on a n by m initial configuration?
- A game playing program often requires a “next configurations” operation that, given a configuration X , produces a list of all configurations that can be reached from X in just one move. How would you implement this operation for Chomp?

Comment

Part 1 is a bit of a give-away. You can find the answer on the web and/or in standard texts, but try to do this on your own, it’s not too hard. For part 3 think about a good, simple data structure for a Chomp configuration.

By the way, don’t try to find a winning strategy, Chomp is very difficult. The 1 by m case is straightforward, but even the 2 by m case is not trivial. Never mind higher-dimensional versions, or versions where infinitely large chocolate bars are considered.

4.12 Necklaces

Background

A necklace is a circular string of colored beads. Two necklaces are considered to be the same if we can rotate one to obtain the other, but reflections are not allowed here (as opposed to bracelets where reflections are allowed). There are two parameters: the length n of the necklace and the number k of colors, correspondingly we speak of (k, n) necklaces.

Task

- A. Compute the number of binary necklaces of length 8.
- B. How many of these necklaces have exactly 4 black and 4 white beads?
- C. Compute the pattern inventory for these necklaces.
- D. Find a general description for the number of (k, n) necklaces.

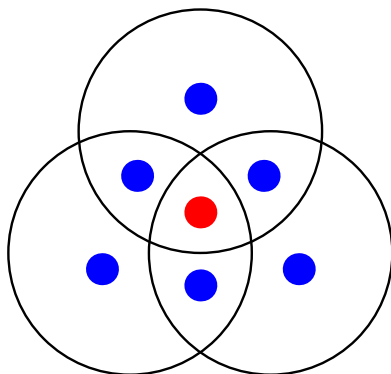
Comment

Try to make the formula in part D as nice as possible (there will not be a really simple closed form).

4.13 Flipping Pebbles

Task

Suppose you have 7 pebbles, blue on one side, and red on the other. They are arranged in three overlapping circles as in the picture below.



Initially, all pebbles are blue side up. You are allowed to flip pebbles, but not individually: you must select a circle, and then flip all pebbles within that circle. The only other permissible operation is to set all pebbles in one circle to red side up. The goal configuration is shown in the picture: the central pebble is flipped over to red, with all the other pebbles still blue (download the pdf from the web so you can see the colors).

Hence there are really 6 operations. Let's agree (for the sake of TA sanity) to call them a, b, c for flipping the pebbles and A, B, C for setting to red. Define a pattern to be an equivalence class under the following equivalence relation: X is equivalent to Y if X can be transformed to Y by a sequence of these operation and Y can be transformed to X .

Task

- A. What part of this game fits within the Burnside/Polya framework? What's wrong with the other part?
- B. How many patterns are there?
- C. Can the target configuration be reached? Justify your answer.

Extra Credit: Find a decent characterization of the reachable configurations.

Comment

Since there are only 128 configurations you can compute this to death – but try to argue without brute-force computation. For example, to settle the feasibility question an invariant might be useful: the operations don't change a certain property, the initial configuration has the property, but the final one does not. Of course, a little brute-force computation may be helpful to sharpen one's "intuition."

4.14 Shuffle

Background

Perfect shuffle is an operation known to all card sharks. Start with a even-sized deck of cards and cut it into two parts of equal size. Then interleave the cards, alternating between both halves. There are two versions, called in-shuffle and out-shuffle, depending on whether the first card is chosen from the right or left half. We will write σ_n for in-shuffle and τ_n for out-shuffle. For example, for $n = 12$ we have

$$\begin{aligned}\sigma_{12} &= (7, 1, 8, 2, 9, 3, 10, 4, 11, 5, 12, 6) \\ \tau_{12} &= (1, 7, 2, 8, 3, 9, 4, 10, 5, 11, 6, 12)\end{aligned}$$

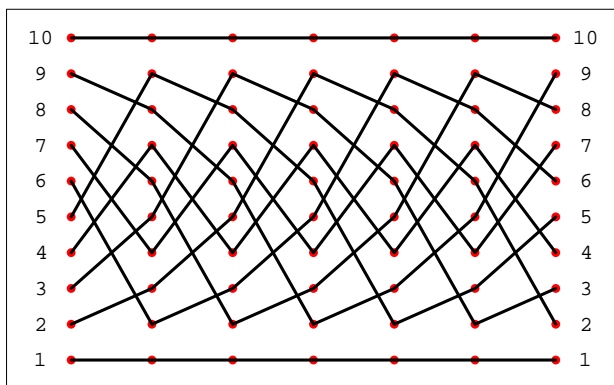
We will always assume that n is even in this problem. Both permutations naturally act on $[n]$, so they can be thought of as acting on a deck of n cards.

Incidentally, if we use ordinary riffle shuffle (where the interlacing is not constrained to exactly one card from each deck), then 7 shuffles suffice to randomize a standard 52 card deck.

Task

1. Let $T_n \subseteq \mathbb{S}_n$ be the group generated by τ_n . What is the size of T_n ? In other words, what is the order of τ_n in \mathbb{S}_n ? Give as nice a description as you can, but don't expect anything as simple as, say, a polynomial in n .
2. Let $S_n \subseteq \mathbb{S}_n$ be the group generated by σ_n . What is the size of S_n ? Use the results for τ , the two permutations are obviously closely related.
3. Is repeated out-shuffle/in-shuffle a good way to randomize a deck of 52 cards? Of the two, which is worse?
4. Assume we have a deck of 10 cards and we identify cards of the same suit. So, we really have a list of length 10 with entries in $\clubsuit, \diamond, \heartsuit, \spadesuit$. How many patterns are there if we identify two decks that can be obtained from each other by out-shuffle?

For the last problem, you might find the following plot of τ very helpful. Trace a few of the orbits and you will see.



4.15 The 15 Puzzle

Background

You are probably all familiar with the old 15 Puzzle: a 4 by 4 square is subdivided into unit squares, and one of those is missing. The only admissible operation is to move an adjacent square into the gap. The squares are numbered 1 through 15, and are initially scrambled. The goal is to bring them into the standard, ordered arrangement:

$$C_0 = \begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & \blacksquare \\ \hline \end{array}$$

Task

- Does this puzzle fit into our framework of group actions? Explain your answer carefully.
- Prove that the configuration C_1 below cannot be transformed to C_0 by any sequence of legal moves.

$$C_1 = \begin{array}{|c|c|c|c|} \hline 2 & 1 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & \blacksquare \\ \hline \end{array}$$

- Classify the configurations of this puzzle with respect to solvability.

Comment

Think about inversions as in the analysis of primitive sorting algorithms.

4.16 Reversible Gates

Background

In this problem we will systematically and criminally confuse n -bit gates and the functions $\mathbf{2}^n \rightarrow \mathbf{2}^n$ they determine. It's not worth the effort to keep things neat and orderly.

In class we claimed that all reversible 2-bit gates are affine maps in the sense that

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = A \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix}$$

where A is one of the following 6 matrices, a and b are constant and all the arithmetic is modulo 2.

$$\begin{array}{ccccccccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

For example, $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $a = b = 0$ simply switches x and y .

Also recall that the 3-bit Toffoli gate has the truth table

x	y	z	x'	y'	z'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Hence $z' = z \oplus (x \wedge y)$, and the other inputs are simply passed through.

Task

1. Verify that all the 2-bit circuits above are indeed reversible.
2. Show that all reversible 2-bit circuits are of this special form.
3. Show that the Toffoli gate can not be constructed from reversible 2-bit circuits.

Comment

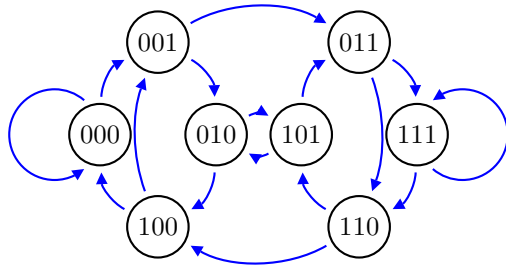
Note that the analogous result for irreversible circuits is plain false: every 3-bit circuit can be constructed from, say, NAND gates.

4.17 Hamiltonian Sequences

Background

A de Bruijn sequence of order k is a bit-sequence H of length $2^k + k - 1$ that contains every bit-sequence of length k as a factor. E.g., $H = 0001110100$ is a de Bruijn sequence of order 3. One way of generating

such sequences is to construct a de Bruijn graph B_k of order k , and then to trace a Hamiltonian cycle in B_k (and flattening out the node sequence to a bit-sequence). Recall that a Hamiltonian cycle is a cycle that uses every node exactly once. Here is a picture of B_3 .



A useful idea in this context is that of an Eulerian cycle: a cycle that uses every edge exactly once (edge, not vertex!). It is easy to see that a directed graph is Eulerian if and only if it is connected and every node has the same in-degree and out-degree.

Task

1. What is the relationship between Eulerian and Hamiltonian cycles for the family of de Bruijn graphs?
2. Explain why these graphs are Hamiltonian.
3. Find a linear time algorithm to construct an Eulerian cycle.
4. Now explain how to construct a de Bruijn sequence of order k , for any k .
5. What is the running time of your algorithm as a function of k ? What is the memory requirement? You have to account for the cost of constructing the graph, as well as the cost of finding the cycle.

Comment

Testing whether a general directed graph is Hamiltonian is very difficult, no polynomial time algorithm is known (nor is one likely to exist). Eulerianess (Eulericity?) is very easy to check on the other hand.

For the Eulerian cycle algorithm you may assume that the graph is given in adjacency list form. The running time of your Eulerian cycle algorithm must be $O(n + e)$, the size of the adjacency list, where n is the number of nodes of the graph, and e the number of edges.

The standard graph-theory argument that one can simply remove a cycle, assume inductively that one already has an Eulerian cycle for the remainder of the graph, and then glue the two cycles together is not good enough algorithmically: you have to explain exactly how to do this in linear time.

4.18 Boolean Circuits

Background

We have seen how to count Boolean functions modulo the equivalence relation induced by inverting some inputs. Several other modifications produce similar classifications of Boolean functions on k inputs. The space of configurations here is

$$X = 2^k \rightarrow 2$$

Task

1. Count the number of Boolean functions when inputs and the output can be inverted.
2. Count when inputs can be rotated. So if the rotation is by one place we have

$$f(x_1, x_2, \dots, x_k) = g(x_2, x_3, \dots, x_k, x_1)$$

3. Count when inputs can be arbitrarily permuted.

$$f(x_1, x_2, \dots, x_k) = g(x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(k)})$$

Comment

Always explain clearly what exactly the group is that acts on the space of Boolean functions.

Try to come up with a concise, elegant answer for the count but don't necessarily hope for a simple closed-form solution.

4.19 Four Fours

Background

There are many problems of the type “given such-and-such arithmetic operations and numbers such-and-such, show how to get value this-and-that”. For the most part, these puzzles are quite boring, but they provide an excellent excuse to implement some algorithms that manipulate and evaluate arithmetic expressions.

For the following problem the admissible operations are plus, minus, times, divide, exponentiate, factorials and square roots. Admissible constants are 4, 4, 4, 4; each of the four instances of 4 must be used. The goal is to represent all positive integers up to 100.

For example, $3 = (4 + 4 + 4)/4$ and $63 = (4^4 - 4)/4$.

Task

- A. How many integers can be written using only the five binary operators?
- B. Add the two unary operators; how many integers can be expressed now?
- C. Lastly, add the two hoaky operations $.4$ (put a decimal point in front of a 4) and $\bar{4} = 0.4444\dots = 4/9$. Can you now obtain all numbers up to 100?

Comment

Try to find an elegant approach, avoiding unnecessarily large searches. Some people have driven the search for four-fours expressions much further, and are allowing absurdly complicated functions in the process.

4.20 Rigid Words

Background

For the sake of simplicity consider only words over the alphabet $\{0,1\}$. For any word x write \bar{x} for the bit-wise complement of x . Recall that a word u is imprimitive if it occurs as a factor of uu in a non-trivial manner. Now define a word u to be rigid if u and \bar{u} appear only in the obvious places in $u\bar{u}$. For example, of all 3-letter words only 010 and 101 fail to be rigid. All words of length 4 are rigid. The next table shows the number of non-rigid words up to length 16.

1	0	9	8
2	0	10	4
3	2	11	2
4	0	12	16
5	2	13	2
6	4	14	4
7	2	15	38
8	0	16	0

This looks rather complicated, one should not expect any easy characterizations.

Task

- A. Extract a conjecture about the number of non-rigid words from the table and prove it.

Comment

As a warm-up exercise you might want to take another look at the proof of the observation that if u is a proper factor of uu then u must be imprimitive.

4.21 Keane Products

Background

For the sake of simplicity consider only words over the alphabet $\{0,1\}$. For any word x write \bar{x} for the bit-wise complement of x . Define the Keane product on binary words as follows.

$$\begin{aligned}u \times \varepsilon &= \varepsilon \\u \times v0 &= (u \times v)u \\u \times v1 &= (u \times v)\bar{u}\end{aligned}$$

For example, $001 \times 10 = 110001$. Since Keane multiplication is associative (see part (A) below) we can form arbitrary finite products without worrying about parentheses. How about infinite products, say

$$v_0 \times v_1 \times v_2 \times v_3 \times \dots$$

In general, one cannot make sense out of this product since the initial segments $V_n = v_0 \times v_1 \dots v_n$ fail to converge. However, if all $v_i, i > 0$, start with a 0 (and $v_0 \neq \varepsilon$) then V_n is a proper prefix of V_{n+1} and the sequence (V_n) has a natural limit $V \in \mathbf{2}^\omega$. This construction is interesting even $v_i = v$ for all i . For example,

$$\begin{aligned} u^\omega &= u \times 0 \times 0 \times 0 \dots \\ (10)^\omega &= 1 \times 010 \times 010 \times 010 \dots \\ 011010011001011010010110011 \dots &= 01 \times 01 \times 01 \times 01 \dots \\ 001001110001001110110110001 \dots &= 001 \times 001 \times 001 \times 001 \dots \end{aligned}$$

The first and last infinite word are periodic, but the second is the famous Morse-Thue word and has a rather complicate structure (for example, it is cube-free).

Task

- A. Show that multiplication is associative: $u \times (v \times w) = (u \times v) \times w$.
- B. Determine for which multipliers v_i the product word $V \in \mathbf{2}^\omega$ is periodic.

Comment

The second part is hard, but the examples given above are ever so helpful. Assume $V = u^\omega$ where u has minimal length; compare the lengths of u and V_n .

Chapter 5

Finite State Machines

5.1 Primitive Words

Background Let w be a non-empty word. A word z is the *root* of w if $w \in z^*$ but there is no shorter word with this property. A word is *primitive* if it is its own root.

For example, ab is the root of $abababab$. The primitive words of length 3 are $aab, aba, abb, baa, bab, bba$.

Task

- Show that any two non-empty words u, v that commute (i.e., $uv = vu$) have the same root.
- Show that a non-empty word w is primitive if, and only if, $w^2 = xwy$ implies that $x = \varepsilon$ or $y = \varepsilon$.
- Count the number of primitive words of length n over a k letter alphabet.
- As part of some algorithm it is necessary to store a few thousand binary words of length 20. Prof. Dr. Wurzelbrunft suggests to speed things up by storing the roots of words, rather than the words directly. What professional advice can you give Wurzelbrunft?

Comment

5.2 Dense Languages

Background

Write $x < y$ for x is lexicographically less than y where $x, y \in \Sigma^*$. A language $L \subseteq \Sigma^*$ is *dense* if

$$\forall x, y \in L (x < y \Rightarrow \exists z \in L (x < z < y))$$

For example, the language $L = (a + c)^*b$ is dense. For words in L up to length 5 lexicographic order looks like so:

$$\begin{aligned} &aaaab, aaab, aaacb, aab, aacab, aacb, aaccb, ab, acaab, acab, acacb, acb, accab, accb, acccb, b, \\ &caaab, caab, caacb, cab, cacab, cacb, caccb, cb, ccaab, ccab, ccacb, ccb, cccab, cccb, ccccb \quad (5.1) \end{aligned}$$

Task

- A. Show that the binary language $L = (a + c)^*b$ is dense. In fact, L is order-isomorphic to the rationals.
- B. Show that for all words $u < v < w$ the language $L_{u,v,w} = (u + w)^*v$ is dense.
- C. Find a way to test whether the language of a given DFA is dense.
- D. What is the complexity of your decision algorithm? Assume the alphabet is fixed.

Comment

5.3 Well-Ordered Languages

Background

Write $x < y$ for x is lexicographically less than y where $x, y \in \Sigma^*$. Consider a language $L \subseteq \Sigma^*$; if $\langle L, < \rangle$ is a well-order we will simply call L *well-ordered* and omit any reference to lexicographic ordering. For alphabets of size larger than one Σ^* is not well-ordered:

$$1 > 01 > 001 > 0001 > 00001 > \dots$$

On the other hand, 1^*0 is easily seen to be well-ordered. We are mostly interested in the case where L is regular.

For well-ordered L we write $\text{ot}(L)$ for its order type. Recall that a language L is called *prefix* if no proper prefix of any word in L is in L . Also, $L \subseteq \mathbf{2}^*$ is called *complete prefix* if it is prefix and for any prefix u of a word in L we have: $u0 \in L \iff u1 \in L$.

Task

- A. Show that the order type of 1^*0 is ω . and the order type of 1^*01^* is ω^2 .
- B. Show that for any well-ordered regular language L there exists another well-ordered regular language L' such that
 - L' is complete prefix over the alphabet $\mathbf{2} = \{0, 1\}$, and
 - the ordertype of L' is the same as the ordertype of L .
- C. Explain how to compute in polynomial time a DFA for L' given a DFA for L .

Comment

For the second part break things up into several steps: prefix, binary and lastly complete (while always preserving ordertype).

5.4 Word Periods

Background

A positive integer p is a *period* of a non-empty word $w = w_1w_2 \dots w_n$ if for all $i \leq n - p$ we have $w_{p+i} = w_i$. The least such p is referred to as *the period* of w .

Let u be primitive and w arbitrary. The u -stable normal form of w is a decomposition of w defined as follows. If u^2 is not a factor, the decomposition is just w . Otherwise we write

$$w = v_0 u^{\alpha_1} v_1 u^{\alpha_2} \dots u^{\alpha_k} v_k$$

where $k \geq 1$ is minimal subject to the following constraints: the v_i must not contain a factor u^2 and $v_0 \in \Sigma^* u$, $v_i \in (\Sigma^* u \cap u \Sigma^*)$ for $1 \leq i < k$ and $v_k \in u \Sigma^*$.

For example, for $u = aba$ and $w = ab(aba)^5 ba(aba)^4 ba$ we have $w = ababa(aba)^3 ababa(aba)^3 ababa$.

Task

- A. Let p be the period of w and u the prefix of w of length p . Show that $w = u^k u_-$ for some $k \geq 1$ and u_- a proper prefix of u .
- B. Again let p be the period of w and u the prefix of w of length p . Show that w is primitive if, and only if, $w \notin uu^*$.
- C. Show that the u -stable normal form exists and is unique.

Comment

The u -stable normal form may seem a bit strange; it turns out to be very useful in the discussion of word equations.

5.5 Combinatorics on Words

Background

Two non-empty words u and v are *conjugate* if there are words x and y such that $u = xy$ and $v = yx$. In other words, v can be obtained from u by a cyclic shift.

Task

- A. Show that a non-empty words u and v are conjugate if, and only if, u is a factor of v^2 .
- B.

Comment

5.6 Converting Regular Expressions

Background

We have seen two methods for converting regular expressions to finite state machines, one due to Thompson and the other to McNaughton and Yamada.

Task

A. Convert the following rational expressions to NFAs, using both the Thompson and the McNaughton-Yamada approach for each expression. You might wish to recycle some of the component machines. Which method yields better results?

- $aaaab^*$,
- $(a + b)^3$,
- $(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$.

B. Convert the NFAs to DFAs. Which machines are smaller?

Comment

5.7 Regularity and Language Operations

Background

We have seen that regular languages are closed under the operations of union, concatenation and Kleene star. Going in the opposite direction is a bit more complicated.

Task

Determine whether the following propositions are true or false. Provide a clear proof for your claim in each case.

- A. $L_1 \cup L_2$ regular and $|L_1| = 1$ implies L_2 regular.
- B. $L_1 \cup L_2$ regular and L_1 finite implies L_2 regular.
- C. $L_1 \cup L_2$ regular and L_1 regular implies L_2 regular.
- D. $L_1 \cdot L_2$ regular and $|L_1| = 1$ implies L_2 regular.
- E. $L_1 \cdot L_2$ regular and L_1 finite implies L_2 regular.
- F. $L_1 \cdot L_2$ regular and L_1 regular implies L_2 regular.
- G. L^* regular implies L regular.

Comment

5.8 Regularity and Palindromes

Background

Task

Which of the following languages is regular over the alphabet $\{a, b\}$? Justify your answer.

- A. $L_1 = \{x \in \{a, b\}^* \mid x = x^r\}$
- B. $L_2 = \{xwx^r \mid x, w \in \{a, b\}^+\}$
- C. $L_3 = \{xx^r w \mid x, w \in \{a, b\}^+\}$

Comment

5.9 Word Binomials

Background

By a subsequence of a word $v = v_1v_2 \dots v_m$ we mean any word $u = v_{i_1}v_{i_2} \dots v_{i_r}$ where $1 \leq i_1 < i_2 < \dots < i_r \leq m$ is a strictly increasing sequence of indices. Thus bbc and cab are subsequences of $ababacaba$ but ccb is not.

Note that a specific word can occur multiple times as a subsequence of another. For example, aab appears 7 times in $ababacaba$. We write

$$\binom{v}{u} = \text{number of occurrences of } u \text{ as a subsequence of } v.$$

The notation is justified since “word binomials” generalize ordinary binomial coefficients: $\binom{n}{k} = \binom{a^n}{a^k}$.

Task

Let $\delta_{a,b} = 1$ iff $a = b$, 0 otherwise, and $a, b \in \Sigma$ and $u, v, u_i, v_i \in \Sigma^*$.

- A. Show that

$$\binom{va}{ub} = \binom{v}{ub} + \delta_{a,b} \binom{v}{u}$$

- B. Show that

$$\binom{v_1v_2}{u} = \sum_{u=u_1u_2} \binom{v_1}{u_1} \binom{v_2}{u_2}$$

- C. Give an efficient algorithm to compute word binomials.
- D. Given a word u and an integer r construct a DFA that accepts

$$L(u, r) = \{v \in \Sigma^* \mid \binom{v}{u} = r\}$$

Comment

5.10 Constrained Quotients

Background

Recall the left quotient of a language L by a word x , an inverse operation to concatenation:

$$x^{-1}L = \{y \mid xy \in L\}$$

Suppose that L is regular. As we have seen, $x^{-1}L$ is also regular in this case. It is easy to see that

$$P(L) = \{x \mid x^{-1}L \neq \emptyset\}$$

is also regular. In fact, one can constrain the length of a witness $y \in x^{-1}L$ in many ways and still obtain regular languages. So suppose we have a function $f : \mathbb{N} \rightarrow \mathbb{N}$ and define

$$P(L, f) = \{x \mid x^{-1}L \cap \Sigma^{f(|x|)} \neq \emptyset\}$$

Let $t(n) = 2n$ and $e(n) = 2^n$.

Task

- A. Show that $P(L)$ is regular.
- B. Show that $P(L, t)$ is regular.
- C. Show that $P(L, e)$ is regular.
- D. Give a bound on the state complexity of $P(L, t)$ and $P(L, e)$.

Comment

5.11 Word Shuffle

Background

The *shuffle* operation, in symbols \parallel , is a map from $\Sigma^* \times \Sigma^*$ to $\mathfrak{P}(\Sigma^*)$ and is defined by

$$\begin{aligned} \varepsilon \parallel y &= y \parallel \varepsilon = \{y\} \\ xa \parallel yb &= (x \parallel yb) a \cup (xa \parallel y), b. \end{aligned}$$

As usual, we extend to languages by

$$K \parallel L = \bigcup \{x \parallel y \mid x \in K, y \in L\}$$

Shuffle is an important idea in the study of concurrency. If we think of letters as atomic actions then the shuffle of two words describes all possible interleavings of two action sequences. Note that the size of the shuffle product of two words behaves quite erratically. For example, for $x = aabb$ and $y \in \{a, b\}^4$ we get the following cardinalities:

aaaa	15	baaa	30
aaab	10	baab	18
aaba	12	baba	23
aabb	6	babb	12
abaa	19	bbaa	36
abab	11	bbab	19
abba	18	bbba	30
abbb	10	bbbb	15

Task

- Given two words x and y with disjoint alphabets, construct a DFA that recognizes $x \parallel y$.
- What is the connection between your DFA and binomial coefficients?
- Given two regular languages K and L , construct a finite state machine that recognizes $K \parallel L$. Assume the languages are represented by DFAs.

Comment

5.12 Shuffle Language

Background

Fix a regular language $L \subseteq \Sigma^*$ and a word $u \in \Sigma^*$. Let

$$\widehat{L : u} = \{x \in \Sigma^* \mid x \parallel u \cap L \neq \emptyset\}$$

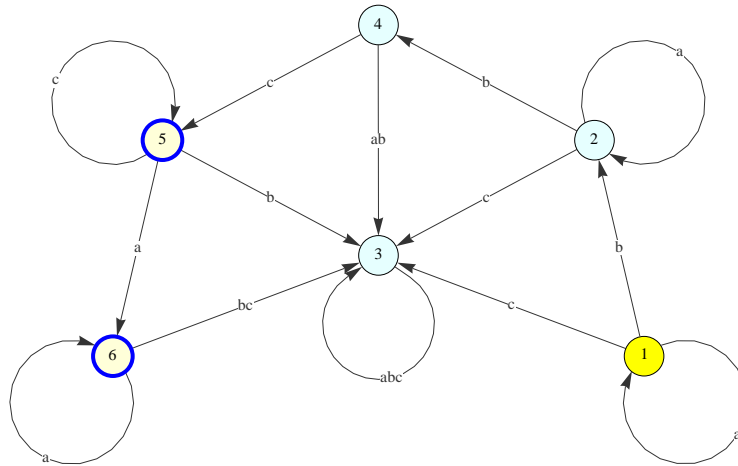
be the collection of all words that can be interspersed with the letters in u to produce a word in L .

Task

- By hand, determine $\widehat{L : u}$ where $L = a^*ba^*c^+a^*$ and $u = bc$.
- Construct the corresponding minimal DFA for $\widehat{L : u}$ as above.
- Give a general construction of a finite state machine for $\widehat{L : u}$ assuming L is represented by a DFA.
- From your construction of an automaton for $\widehat{L : u}$, derive a bound on the state complexity of this language.

Comment

For the last part an automaton with ε -transitions is perfectly acceptable. The minimal DFA for the language L in parts A and B looks like this:



5.13 Frontiers

Background

Recall that a language is prefix if no word is a prefix of another word in the language. There is a close connection between prefix sets and prefix-closed sets. To see this, define the *proper prefixes* and the *frontier* of a language $L \subseteq \Sigma^*$ as follows.

$$\begin{aligned} \text{pp}(L) &= \Sigma^* - L \cdot \Sigma^* \\ \text{fr}(L) &= L \cdot \Sigma - L \end{aligned}$$

Clearly, $\text{pp}(L)$ is prefix-closed and $\text{fr}(L)$ is prefix. $L \subseteq \Sigma^+$ is *right complete* if for any word w there exists a word $u \in L$ such that $w\Sigma^* \cap u\Sigma^* \neq \emptyset$.

Task

- Show that $\text{pp}(\text{fr}(C)) = C$ whenever $C \subseteq \Sigma^*$ is prefix-closed.
- Show that $\text{fr}(\text{pp}(P)) = P$ whenever $P \subseteq \Sigma^*$ is prefix.
- Show that L is a frontier language if every for every word w there is some $x \in L$ such that $w \leq x$ or $x < w$.

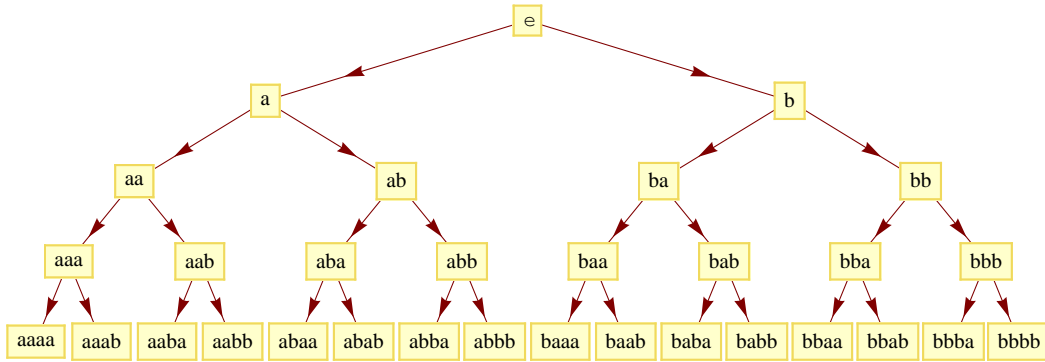
Comment

Frontier languages are useful in constructing DFAs for regular languages by unfolding the minimal DFA.

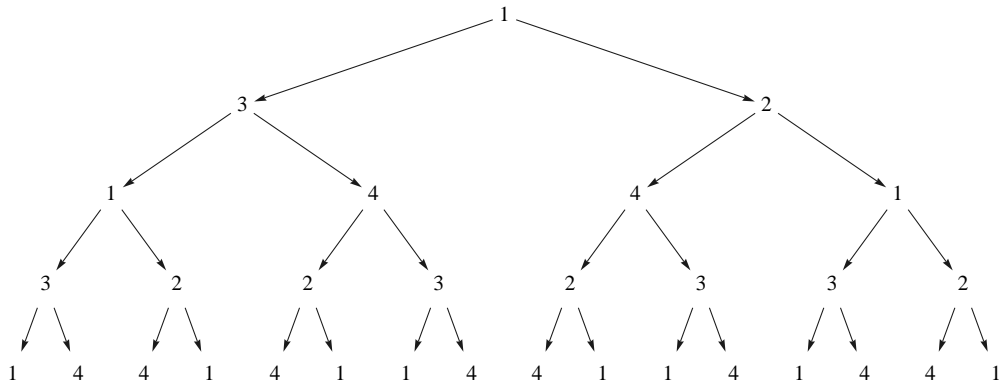
5.14 Frontier Automata

Background

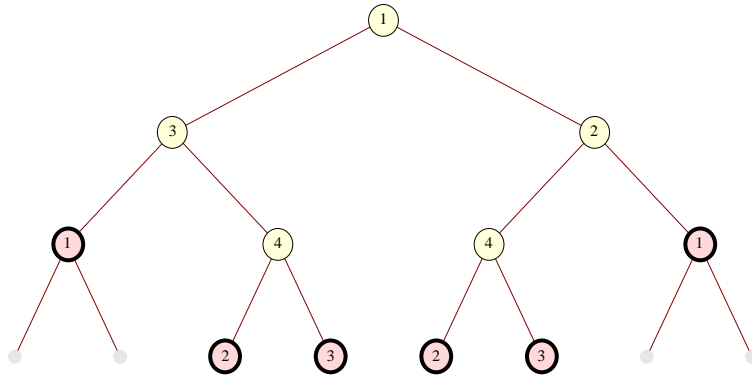
We can think of all words over some alphabet Σ as an infinite tree. For example, the first few layers for the tree of $\Sigma = \{a, b\}$ looks like so:



Now suppose we have a DFA M that accepts some regular language L . We can label the nodes in the tree by the corresponding states of M : node w is labeled by $\lambda(w) = \delta(q_0, w)$. For example, for the canonical DFA that checks for even number of a 's and b 's we get



Now suppose X is a finite, maximal-prefix set of words X such as indicated by the red nodes below.



Define a *frontier* for M to be any such set X subject to the condition that $x \in X$ implies $\lambda(x) = \lambda(y)$ for some $y \in \text{pp}(X)$. This condition holds for the example.

Task

- A. Show that for any M there is a frontier with the additional property that $\lambda(x) = \lambda(y)$ for some proper prefix y of x .
- B. Given a frontier for M , explain how to construct a new DFA for L whose states are $\text{pp}(X)$.

5.15 A Groupoid

Background A perennial favorite: problem 2.8 b in Hopcroft and Ullman’s classical text.

Below is the multiplication table for an operation on the set $\Sigma = \{a, b, c\}$.

\circ	a	b	c
a	a	a	c
b	c	a	b
c	b	c	a

The operation fails to be associative, so in order to evaluate products of length larger than 2 we need to specify a particular multiplication order. Two natural choices are left-associative multiplication and right-associative multiplication:

$$\lambda(x_1 x_2 \dots x_n) = (\dots (x_1 \circ x_2) \circ \dots x_{n-1}) \circ x_n$$

$$\rho(x_1 x_2 \dots x_n) = x_1 \circ (x_2 \dots \circ (x_{n-1} \circ x_n) \dots)$$

For completeness, let $\rho(s) = \lambda(s) = s$ for $s \in \Sigma$. If we think of sequences over Σ as words the question arises: what is the complexity of the language

$$A = \{x \in \Sigma^+ \mid \lambda(x) = \rho(x)\}$$

As it turns out, the language is regular, but construction of a corresponding finite state machine is a bit difficult.

Task

- A. Construct a DFA L_s that accepts all non-empty strings x such that $\lambda(x) = s$.
- B. Construct an FA R_s that accepts all non-empty strings x such that $\rho(x) = s$.
- C. Conclude that A is regular.
- D. **Extra Credit:** What is the state complexity of A ?

Comment

The last part requires some amount of brute force computation.

5.16 A Quadratic Language Equation

Background

We have seen that certain linear equations for languages can be solved (Arden's Lemma). In general, though, even simple equations for languages are rather difficult to deal with. For example, consider the problem of solving

$$L \cdot L = \Sigma^*$$

A trivial solution is $L = \Sigma^*$, but how about other solutions? For simplicity consider only the alphabet $\{a, b\}$.

Task

- A. Verify that $L = \varepsilon + a\Sigma^* + \Sigma^*b$ is a solution.
- B. The state complexity of the last solution is 4. Find a solution of state complexity 3.
- C. Find another solution of state complexity 3 that cannot be obtained from the previous one by swapping a and b .

Comment

There are 92 solutions of state complexity 3, so there is quite a bit of choice. Think about excluded words.

5.17 Forbidden Factors

Background

Given a finite collection of words L , the *forbidden factors*, we can form the regular language \widehat{L} of all words not containing a factor in L . Here are two concrete examples

$$L = \{aaa, aba, bab, bbb\}$$

$$K = \{aaa, aba, bbb\}$$

The growth rates of \widehat{L} and \widehat{K} can be seen in the following table:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
\widehat{L}	1	2	4	5	6	7	9	11	13	16	20	24	29	36	44
\widehat{K}	1	2	4	4	4	4	4	4	4	4	4	4	4	4	4

Task

- A. Explain the growth rate of \widehat{K} .
- B. Explain the growth rate of \widehat{L} .

Comment

Do not try to find a closed form for the growth rate of \widehat{L} , argue asymptotically.

5.18 Subwords and Halfs

Background

Write $\text{sw}_\ell(x)$ for the collection of all factors of x of length ℓ . Thus, $\text{sw}_\ell(x)$ has cardinality $|x| - \ell + 1$ for all words x of length at least ℓ , and 0 otherwise.

Task

- A. Let $K \subseteq \Sigma^{\leq \ell}$ be a finite language. Construct a DFA for the language

$$L_{\ell,K} = \{ x \in \Sigma^* \mid \forall u \in \text{sw}_\ell(x)(u \in \Sigma^* K \Sigma^*) \}$$

- B. Let L be any regular language. Construct a finite state machine for the language

$$L_{1/2} = \{ x \in \Sigma^* \mid \exists y \in \Sigma^* (xy \in L \wedge |x| = |y|) \}$$

- C. How about

$$L_{\text{rep}} = \{ zz \in \Sigma^* \mid z \in \Sigma^* \}$$

5.19 The Un-Equal Language

Background

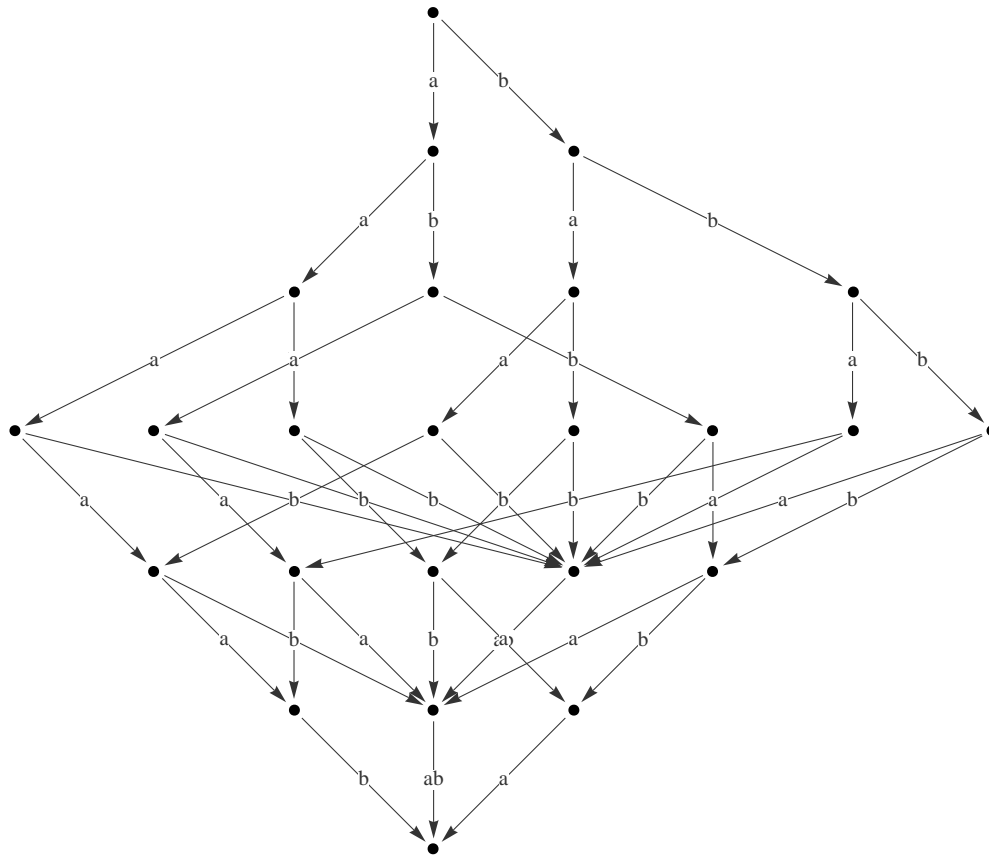
Consider the language of all strings of length $2k$ that are not of the form uu :

$$L_k = \{ uv \in \{a, b\}^* \mid |u| = |v| = k, u \neq v \}.$$

These languages are finite, hence trivially regular. The following table shows the state complexity of L_k up to $k = 6$.

k	1	2	3	4	5	6
sc	5	12	25	50	99	196

The minimal DFA for L_3 looks like so:



The sink has been omitted, the top state is initial and the bottom state final.

Task

- A. Determine all quotients for L_2 .
- B. Generalize. In particular explain the diagram for L_3 .
- C. Determine the state complexity of L_k
- D. Determine the state complexity of $K_k = \{uu \mid u \in \{a, b\}^k\}$.

Comment

From the diagram and the table it is not hard to conjecture a reasonable closed form for the state complexity. For a proof one can exploit the description of the minimal DFA in terms of quotients.

5.20 Representations of Regular Languages

Background

Task

We have seen that a regular language (i.e., a language that is accepted by a finite state machine) can also be described by a regular expression or by a formula in $\text{MSO}[<]$.

Recall that u is a factor of v if $\exists x, y \in \Sigma^* (v = xuy)$ and $u = u_1u_2 \dots u_n$ is a subword of v if $\exists x_0, \dots, x_n \in \Sigma^* (v = x_0u_1x_1u_2x_3 \dots u_nx_n)$.

Let K be the language of all words over alphabet $\Sigma = \{a, b, c\}$ containing exactly two factors ab . Likewise, let L be the language of all words over alphabet $\Sigma = \{a, b, c\}$ containing exactly two subwords ab .

- Construct the minimal automaton for K and L .
- Give a regular expression for K and L .
- Give a formula in $\text{MSO}[<]$ for K and L .

Comment

For the regular expressions and formulae try to find short, elegant answers.

5.21 Primitivity and Minimality

Background

A word u is *primitive* Let w be an arbitrary word. A word z is the *root* of w if $w \in z^*$ but there is no shorter word with this property. A word is *primitive* if it is its own root. For example, ab is the root of $abababab$. The primitive words of length 3 are $aab, aba, abb, baa, bab, bba$.

Given a binary word $u = u_0u_1 \dots u_{n-1}$ define the corresponding subset of $S(u) \subseteq \{0, 1, \dots, n-1\}$ by $i \in S(u) \iff u_i = 1$ (in other words, think of u as a bitvector for membership). Define a DFA $M(u)$ over the one-letter alphabet $\{a\}$ as follows:

$$M(u) = \langle \{0, 1, \dots, n\}, \{a\}, \delta; 0, S(u) \rangle$$

where $n = |u|$ and $\delta(p, a) = p + 1 \pmod n$.

Task

- Show that any two non-empty words u, v commute (i.e., $uv = vu$) have the same root.
- Show that $M(u)$ is minimal if, and only if, u is primitive by reasoning about the behavior of the states in $M(u)$.
- Reprove part (B) by “running” a minimization algorithm on $M(u)$.

Comment

Since minimization algorithms compute behaviors a lawyer might argue that part (C) is also an answer to part (B). Don't even think about it.

5.22 Definite Automata

Background

Finite state machines are often called memoryless devices since they do not have access to any kind of RAM memory. On the other hand, even a DFA can remember certain inputs unboundedly long. Here is a type of finite state machine that truly forgets.

An accessible DFA is *definite* if for some $r \geq 0$, all words of length at least r and all states p and q : $\delta(p, x) = \delta(q, x)$. Thus, in a definite DFA, acceptance of a sufficiently long input depends solely on the suffix of length r of the input. A regular language is definite if it can be accepted by a definite DFA.

Task

- Show that $L_{a,-3}$, the language of all words with an a in position -3 , is definite.
- Let $A \subseteq \Sigma^*$ be any finite language. Show that $\Sigma^* A$ is definite.
- Develop an algorithm that tests if a DFA is definite. Make sure to explain why your algorithm is correct.
- What is the complexity of your algorithm?

Comment

5.23 Right Quotients

Background

Left quotients are defined by removing a prefix from a word. Needless to say, we can also define *right quotients* by removing a suffix:

$$L/x = \{ y \in \Sigma^* \mid yx \in L \}$$

Algebraically, these right quotients behave very much like their left cousins.

Left quotients are directly related to behaviors and can be used to decompose a regular language; one might wonder if regular languages can also be described in some interesting way in terms of right quotients. The answer is yes, but things become a bit more complicated.

In the following, assume M is a minimal DFA accepting some language L . Fix some state p and, for all states $q \neq p$, choose a witness $w_q \in \Sigma^*$ such that $\delta(p, w_q) \in F \iff \delta(q, w_q) \notin F$.

Task

- Show that for any Boolean operation \oplus we have $(L \oplus K)/x = L/x \oplus K/x$.
- Let $K \subseteq L$ be the language accepted by $\langle Q, \Sigma, \delta; q_0, \{p\} \rangle$. Show that $K = \bigcap_{q \neq p} R_q$ where $R_q = L/w_q$ whenever $\delta(p, w_q) \in F$, and $R_q = \Sigma^* - L/w_q$ otherwise ($\delta(p, w_q) \notin F$).
- Write L as a Boolean combination of right quotients of L .

Comment

5.24 Direct Languages

Background

A DFA is *direct* if it has at most one final state. A regular language is direct if it is accepted by some direct DFA. Note the restriction to DFAs; the language $\{a, aa\}$ for example fails to be direct.

Task

- Find a regular language L such that $L - F$ fails to be direct for any finite language F .
- Explain the connection between direct languages and prefix languages.

Comment

5.25 Prefix Languages

Background

A language is *prefix* if no word in the language is a proper prefix of another word in the language. Some would argue that these languages should be called prefix-free, but why should terminology yield to logic?

Task

- Given an algorithm that decides whether the language accepted by a DFA is prefix.
- What is the running time of your algorithm?

Comment

5.26 Recognizing Suffixes

Background

A DFA has no a priori information about the length of an input string. For example, it does not know when it scans the third symbol from the end (as opposed to the third symbol from the start). As a result, it is quite difficult to construct DFAs that recognize languages based on conditions of some suffix of the input.

For any integer $k \neq 0$ write

$$L_{a,k} = \{x \in \{a,b\}^* \mid x_k = a\}$$

For negative k this is interpreted as “ k th symbol from the back”. For example,

$$\begin{aligned} L_{a,2} &= \{aa, ba, aaa, aab, baa, bab, \dots\} \\ L_{a,-3} &= \{aaa, aab, aba, abb, aaaa, aaab, \dots\} \end{aligned}$$

Task

- A. Show how to construct a DFA that accepts $L_{a,k}$ for positive k .
- B. Show how to construct a DFA that accepts $L_{a,k}$ for negative k .

Part 1 is easy, but part 2 requires some work. Try to find an elegant description of these machines.

5.27 Local Languages

Background

Regular languages can be construed as homomorphic images of certain particularly simple regular languages, so called *local* languages. A Σ -language L is local if there are sets $A, B \subseteq \Sigma$ and a binary relation R on Σ such that $L(A, B, R) = \{x_1x_2 \dots x_n \mid x_1 \in A, x_n \in B, R(x_i, x_{i+1})\}$. For example, let $\Sigma = \{a, b, c, d\}$ be an ordered alphabet with order relation $<$ as indicated. Then $L(\{a\}, \{d\}, <) = a^+b^*c^*d^+$, $L(\{a\}, \{a\}, <) = a^+$ and $L(\{d\}, \{a\}, <) = \emptyset$.

Task

- A. Show that every local language is regular.
- B. Show that every regular language is a homomorphic image of a local language.

Comment

5.28 D_4 Recognizer

Background

Consider the dihedral group D_4 with generators a and b where $a^2 = b^4 = 1$ and $ba = a^3b$. If we think of a and b as symbols over a two-letter alphabet words over this alphabet naturally correspond to group elements. More precisely, there is an evaluation map $\eta : \Sigma^* \rightarrow D_4$ which turns out to be a monoid homomorphism.

Task

- A. Show that the language $L \subseteq \Sigma^*$ of all words that evaluate to 1 is regular. Construct the minimal DFA for L .
- B. Determine the cardinality of $L \cap \Sigma^n$.

Comment

5.29 Recognizing Permutations

Background

We can think of a permutation of length k as a word of length k over a k -letter alphabet. E.g., all permutations of length 3 are given by the finite language

$$P_3 = \{abc, acb, bac, bca, cab, cba\}$$

Likewise, we can represent subgroups of the full symmetric group by subsets of P_k . For the alternating group A_4 we obtain

$$P_4^e = \{abcd, acdb, adbc, badc, bcad, bdca, cabd, cbda, cdab, dacb, dbac, dcba\}$$

Task

- A. Determine the state complexity of P_k .
- B. Determine the state complexity of P_k^e .
- C. How about $P_k - P_k^e$?

Comment

The solution for the alternating group requires some special cases for small k .

5.30 Balance and Majority

Background

Write $|x|_0$ for the number of 0's in a binary word x , and likewise $|x|_1$ for the number of 1's. The balance language L_{bal} is the set of all binary words that have the same number of 1's and 0's:

$$L_{bal} = \{x \in \mathbf{2}^* \mid |x|_1 = |x|_0\}.$$

The majority language L_{maj} is the set of all binary words that have at least as many 1's as 0's:

$$L_{maj} = \{x \in \mathbf{2}^* \mid |x|_1 \geq |x|_0\}.$$

Task

- A. Show that there is no finite state machine that accepts L_{bal} .
- B. Show that there is no finite state machine that accepts L_{maj} .

Comment

The essential issue is that in order to recognize these languages a DFA would require some kind of unbounded counting ability – and DFAs can only count in a bounded way. Try to come up with a clean, concise argument. Also note that your argument has to cover all possible finite state machines, you cannot make any assumptions about, say, the size of the machine.

5.31 State Merging

Background

Below is the transition matrix for a 15-state DFA M over the alphabet $\{a, b\}$. Initial state is 1 and the final states are $\{8, 9, 10, 11\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	2	4	6	8	10	12	14	8	10	12	14	8	10	12	14
b	3	5	7	9	11	13	15	9	11	13	15	9	11	13	15

The machine M turns out to be non-minimal.

Task

- Use the state merging algorithm to compute the minimal DFA for this machine.
- What is the language accepted by the machine? Give a short, clear description.
- Try to generalize: the language of M is just one example of a class of languages L_k , $k \geq 1$. What do the minimal automata look like in general? In particular, pin down the state complexity of these languages.

Comment

Make sure to build a nice table for the state merging process, don't just write down the final result. If you like, you can write a program to do this or use my code on the web. It's a good idea, though, to run the algorithm by hand a couple of times, just to get the hang of it.

For part C it's a good idea to draw a nice picture of the minimal automaton for M .

5.32 Minimal Automata for Finite Languages

Background

Brute-force construction of DFAs often lead to machines that are far from minimal. One such example is the application of the product machine construction to obtain a DFA for a finite language such as

$$L = \{ba, bb, aaa, aab\} \subseteq \{a, b\}^*$$

Task

- Construct the accessible part of the brute-force product machine for L , built from the natural DFAs M_w for $w \in L$.
- Construct the minimal DFA for L directly by hand, making sure not to introduce any unnecessary states. You might wish to identify the states with prefixes of the words in L .

- C. Use the minimization algorithm from class to check that your product machine after state-merging produces the same (really: isomorphic) automaton as in part 2.
- D. In a practical algorithm one would not necessarily represent the transition function of the DFA by a two-dimensional lookup table or hashtable. What other data structure lends itself naturally to implement an DFA for a finite language? Explain your choice.

Comment

If you have problems with the construction in part 2, start with a language with just 2 words and figure out how to build the minimal DFA in that case.

5.33 Fast Equivalence Testing

Background

In class we gave an algorithm to test whether two DFAs M_1 and M_2 are equivalent that uses the Union/Find data-structure to maintain an equivalence relation E on $Q = Q_1 \cup Q_2$. The algorithm avoids minimization and runs in essentially linear time.

For this problem consider the following two machines, both over alphabet $\{a, b\}$. M_1 has 4 states, initial state 1 and final states $\{3, 4\}$. The transition matrix is

	1	2	3	4
a	2	3	3	2
b	1	4	4	1

M_2 has 7 states, initial state 1 and final states $\{4, 5\}$. The transition matrix is

	1	2	3	4	5	6	7
a	2	4	6	4	6	4	6
b	3	5	7	5	7	5	7

Make sure to renumber the states in M_2 to $\{5, 6, \dots, 11\}$ to make the state sets disjoint.

Task

- A. As an example, compute the equivalence relation E for the two machines from above. Are they equivalent?
- B. Prove that the algorithm is correct in general. To this end it's probably a good idea to show that $\delta(q_{01}, x) E \delta(q_{02}, x)$ for all words x .

Needless to say you, don't have to use the Union/Find method in your hand calculation. It is much easier to maintain a list of blocks.

5.34 Counting Minimal 1-Letter DFAs

Background

It is rather difficult to determine the number of (n, k) -DFAs that are minimal (up to isomorphism). However, the in special case where $k = 1$ things are manageable. The following table shows the number of minimal $(n, 1)$ -DFAs up to $n = 12$.

n	1	2	3	4	5	6	7	8	9	10	11	12
#	2	4	12	30	78	180	432	978	2220	4926	10908	23790

Task

- Find a way to count the number of minimal $(n, 1)$ -DFAs.
- Generate all minimal $(6, 1)$ -DFAs and explain their acceptance languages.

Comment

Try to exploit part (A) for part (B); your method should work in practice for $n = 10$ as well. Brute force is not a good idea.

5.35 Divisibility In Reverse Binary

Background

We have seen that one can check divisibility by a fixed modulus m on a DFA regardless of the base B . More precisely, we showed how to construct a DFA for words x written in base B notation with the MSD in the first position. Of course, there is no real reason why the MSD should be up front, it might as well be the last digit (reverse binary notation).

Task

- Construct a DFA that tests divisibility by 3 for numbers written in reverse binary. To do this, find a way to express $\nu(xa)$ in terms of $\nu(x)$ and $|x|$ and choose your state set accordingly.
- Show how to generalize this construction for arbitrary moduli and bases.
- How does the size of your machine for $m = 5$ and $B = 2$ in reverse binary compare to the machine constructed in class for normal binary notation? What is going on?

Comment

Naturally one would like to know how large the minimal automata are in general, for arbitrary base B and arbitrary modulus m , and for ordinary base B notation as well as reverse base B . Lots of extra credit if you can give a general characterization.

5.36 Forward State Merging

Background

Below is the transition matrix for a (somewhat random) 13-state DFA M over the alphabet $\{a, b\}$. Initial state is 1 and the final states are $\{4, 6, 8, 9, 10, 11\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
a	2	4	6	8	12	9	12	13	12	13	12	12	13
b	3	5	7	12	10	13	11	12	12	13	12	13	12

This machine turns out to be non-minimal.

Task

- Use the forward state merging algorithm to compute the minimal DFA M_0 for this machine.
- Describe the language L accepted by the machine.
- Construct the minimal DFA M_1 for L directly by hand, using whatever method you prefer.
- Show that M_0 and M_1 are isomorphic.

Comment Make sure to build a nice table for the state merging process, don't just write down the final result. If you like, you can write a program to do this (or use the code on the web).

Part C. will be obvious once you have done part B.

5.37 Determinization

Background

We have seen that nondeterministic machines are often much easier to construct for given languages than their deterministic counterparts. Here are two such examples.

$$A = \{w, w'\}$$
$$B = \{x \in \{a, b\}^* \mid x_{-k} = a\}$$

Here w and w' are arbitrary distinct words over $\{a, b\}$.

Task

- Explain what the natural NFA (no ε -moves) for A looks like. What is the state complexity?
 - Determine the DFA M obtained from this machine by applying the power automaton construction (accessible part only). Give a bound on the state complexity.
 - Should one expect the machine M to be minimal? Explain.
 - Repeat for the language B .
-

5.38 Determinization and Blowup

Background

From an algorithmic perspective, the Rabin-Scott powerset construction has the obvious problem of being potentially exponential, not just in time but even in space.

Task

- A. Explain how to implement the Rabin-Scott determinization procedure, assuming an alphabet of fixed size k . Try to make the running time is nearly linear in the size of the (accessible part of the) output machine as possible.
- B. Construct a family of nondeterministic machines A_n on n states over a 2-symbol alphabet such that the power automaton of A_n has size 2^n and is already minimal.

Comment

Think about what it means for the states of a power automaton (i.e., subsets of the state set of the nondeterministic machine) to be behaviorally inequivalent.

5.39 Universal Finite Automata

Background

Our definition of acceptance of a nondeterministic machine involves existential quantification: there has to be at least one accepting computation. Suppose we modify the definitions so that all computations are required to be accepting. More precisely, let M be an ε -free machine. Define the following acceptance predicate where τ denotes the transition relation of M .

$$\begin{aligned}\text{accept}(p, \varepsilon) &= p \in F \\ \text{accept}(p, az) &= \forall q (\tau(p, a, q) \rightarrow \text{accept}(q, z))\end{aligned}$$

A word x is accepted by M if $\text{accept}(p, x)$ holds for all initial states p . Let us refer to these machines as *universal NFAs*.

Task

- A. If M is a DFA, then this notion of acceptance coincides with the classical definition of acceptance.
- B. Find examples where a universal machine is smaller than its traditional counterpart.
- C. Show that every universal machine can be converted into a traditional finite state machine. What is the size of the deterministic machine?

Comment

5.40 The Dyck Language

Background

Here is a formal description (actually, several) of the language of all balanced strings of parentheses, the so-called Dyck language D . Thus, $[[[]][[]]]$ is a string in D but $[[[]]$ and $][$ are not. Less informally, we can define D by a closure condition as follows. D is the least collection L of strings over alphabet $\Sigma = \{[,]\}$ that

- L contains ε ,
- whenever x, y are in L so is $[x]$ and xy .

Here is an alternative definition of D . First define a binary relation ρ on Σ^* as follows.

$$x \rho y \iff \exists u, v \in \Sigma^* (x = u[]v \wedge y = uv).$$

Let ρ^* be the transitive, reflexive closure of ρ . Yet another approach to D is in terms of a bookkeeping function. Let us define a map $f : \Sigma^* \rightarrow \mathbb{Z}$ by

$$f(x) = |x|_] - |x|_[.$$

Task

- Show that repeated removal of adjacent pairs of matching parens leads to the empty string: $D = \{x \in \Sigma^* \mid x\rho^*\varepsilon\}$.
- Show that $x \in D \iff f(x) = 0 \wedge \forall z \in \text{Pref}(x) (f(z) \geq 0)$

Comment

You should think about generalizations to languages involving several types of parens.

5.41 Pumping

Background

Call a language L *pumpable* if there exists a number N (which depends solely on L) such that for all words w in L with $|w| \geq N$, there exist words x, y and z such that:

- $w = xyz$, $|xy| \leq N$ and $y \neq \varepsilon$,
- for all $t \geq 0$: $xy^tz \in L$.

The famous Pumping Lemma can then be stated like so: Every regular language is pumpable.

Task

A. Use the Pumping Lemma to show that the following languages are not regular.

$$L_1 = \{ a^i b^i \mid i \geq 0 \}$$

$$L_2 = \{ x \in \{a, b\}^* \mid |x|_a = |x|_b \}$$

$$L_3 = \{ 0^{2i} 1^{3i} \mid i \geq 0 \}$$

B. Show that the following language is pumpable but fails to be regular:

$$L = \{ x \in \{a, b\}^* \mid x \text{ contains a subword } b^2 \text{ or } |x|_b \text{ is a square} \}.$$

Comment

5.42 DFAs versus Regular Expressions

Background

Regular expressions are the preferred representation for regular languages in many interactive applications. For example, text editors such as `emacs` use regular expressions for search and replace operations. Clearly, it would be unreasonable to expect the user to type in a corresponding finite state machine in this case. Alas, sometimes machines, even deterministic ones, are a better representation.

In the following, let L_{ee} be the language of all strings over $\{a, b\}$ that have an even number of a 's and an even number of b 's. Likewise, let L_{oo} be the odd/odd language.

Task

- Construct the minimal DFAs for L_{ee} and L_{oo} .
- Give a reasonably simple regular expression for L_{ee} .
- Give a regular expression for L_{oo} .

Comment

The second part is not too bad, but the third is already a bit messy. Never mind the obvious generalizations.

5.43 Hard Regular Expressions

Background

Every regular expression corresponds to a DFA and conversely. However, there is no simple relationship between the sizes of the automata and the regular expressions.

For the following problem, define

$$L_{i,j} = \{ x \in \{a, b\}^* \mid |x|_a = i \pmod{2}, |x|_b = j \pmod{2} \}$$

The minimal DFA for these languages is trivial:

Task

- A. Find a regular expression for L_{00} .
- B. Find a regular expression for L_{11} .
- C. Find a regular expression for L_{01} .

Comment

You must explain why your expressions are correct.

5.44 ω -Regular Languages

Background

$$L = \bigcup W_{q_0,p} W_{p,p}^\omega$$

An ω -word is ultimately periodic if it is of the form uv^ω where $u, v \in \Sigma^*$.

Task

- A. Show that the language of all ultimately periodic words is not regular.
- B. something else

Comment

You must explain why your expressions are correct.

5.45 Acceptance for Büchi Automata

Background

We have seen that acceptance of a Büchi automaton is undecidable if we consider sufficiently complicated input words (e.g., words specified by a computable function). However, for sufficiently simple words acceptance is still decidable.

Let $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ be a Büchi automaton.

Task

- A. Show how to decide whether \mathcal{A} accepts a “constant” input word $U = a^\omega$ where $a \in \Sigma$.
- B. Generalize to periodic inputs $U = u^\omega$ where $u \in \Sigma^*$.
- C. Generalize to ultimately periodic inputs $U = vu^\omega$ where $u, v \in \Sigma^*$.
- D. What is the running time of your three algorithms?

Comment

One might try to push things a bit further and consider inputs that are not periodic but have some reasonably simple description. For example, define

$$U(k) = \begin{cases} 1 & \text{if } k \text{ is prime,} \\ 0 & \text{otherwise.} \end{cases}$$

So

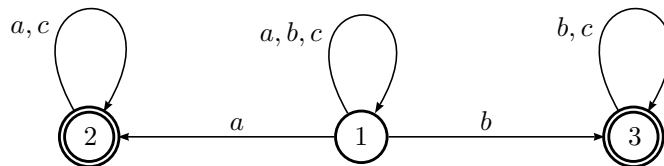
$$U = 0011010100010100010100010000010100000100010100010000010000010\dots$$

Do you feel it is decidable whether some arbitrary Büchi automaton accepts U ?

5.46 Determinization of Büchi Automata

Background

Below is the natural Büchi automaton \mathcal{A} for the language $L \subseteq \{a, b, c\}^\omega$ of all words that contain either at least one a but finitely many b 's or at least one b but finitely many a 's (look at the machine if this makes no sense).



Here $I = \{1\}$ and $F = \{2, 3\}$.

Task

- Write a “regular expression” for this language.
- Run Safra’s algorithm on \mathcal{A} to obtain a Rabin automaton for L .
- Construct a Büchi automaton for the complement of L .

Comment

For the last part, make sure to draw a picture. It’s not at all bad if you pick the right layout. Step 1 is to draw a nice picture for the Rabin automaton in part (B). Think linear.

Solution: Determinization of Büchi Automata

Part : Regular Expression

In terms of counting letters the language can be described as

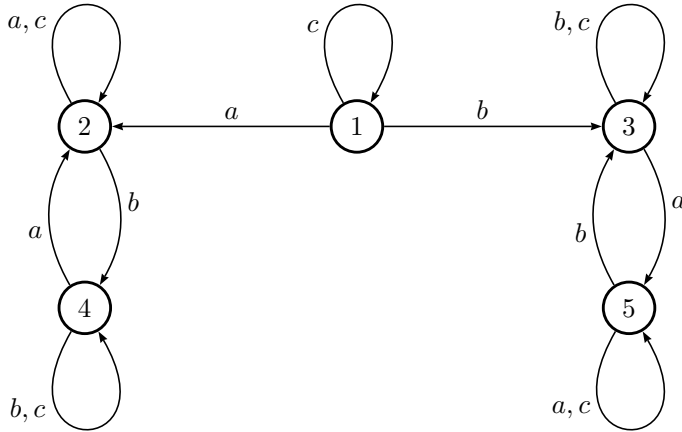
$$(\#_a \geq 1 \wedge \#_b < \infty) \vee (\#_b \geq 1 \wedge \#_a < \infty)$$

This translates into the following regular expression:

$$(a + b + c)^*(a(a + c)^\omega + b(b + c)^\omega)$$

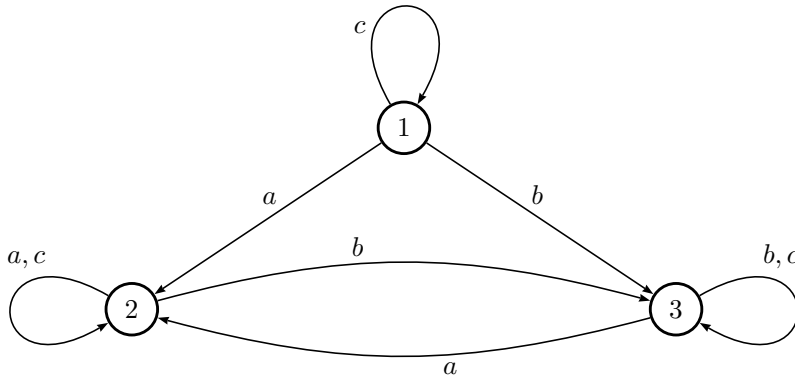
Part A: Safra

The Safra automaton has 5 states (using the version that updates state sets before branching; the other version produces 7 states):



The Rabin pairs are $(1, 4, 5; 2, 3)$ and $(1, 2, 3; 4, 5)$.

Note that there is a high degree of symmetry in this automaton. In fact, we can simplify the machine a bit by merging states in the component on the left with corresponding states in the component on the right.



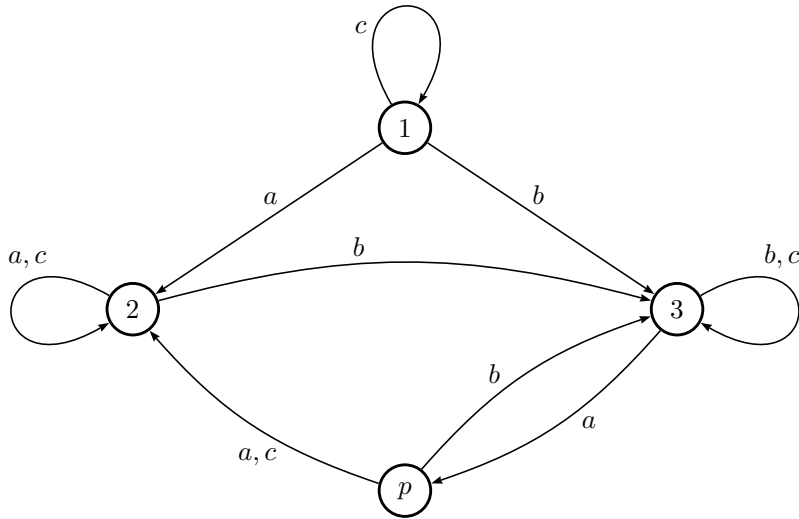
The new Rabin pairs are $(1, 2; 3)$ and $(1, 3; 2)$.

Part B: Complement Büchi

For complementation note that the Muller table for the machine from the last section is $\{2\}, \{3\}$. For the complement the table, after some cleanup based on the structure of strongly connected components of the machine, we get $\{1\}, \{2, 3\}$.

Unfortunately, this does not directly translate into a Büchi automaton. In the interesting case, we need to touch both 2 and 3 infinitely often. To enforce this via a Büchi acceptance condition we need to rewire the

2-point strongly connected component. Here is the modified automaton:



Note that the resulting Büchi automaton is still deterministic; the set of final states is $\{1, p\}$, corresponding to the two components that make up the complement of the language: $c^\omega \cup ((a + b + c)^* a (a + b + c)^* b)^\omega$. Or, in terms of counting:

$$(\#_a = 0 \wedge \#_b = 0) \vee (\#_a = \infty \wedge \#_b = \infty)$$

5.47 Counting Letters

Background

One can simplify formal languages by pretending that letters commute: $ab = ba$ and so forth. A word w can then be described by its *Parikh vector*: let $\Sigma = \{a_1, \dots, a_k\}$ and define

$$\#w = (e_1, \dots, e_k) \text{ where } e_i = |w|_{a_i}$$

so that $\# : \Sigma^* \rightarrow \mathbb{N}^k$.

It is tempting to define $\#L = \{\#x \mid x \in L\} \subseteq \mathbb{N}^k$ for a language L but for our purposes it is more convenient to think of $\#L$ as a language over Σ , the *commutative image* or *Parikh language* of L , defined by

$$L^\# = \{a_1^{e_1} \dots a_n^{e_n} \mid \exists x \in L e_i = |x|_{a_i}, i = 1, \dots, k\}.$$

Task

- A. Show that the Parikh language $L^\#$ is context sensitive whenever L is context sensitive.
- B. Show that there exists a regular language R such that $R^\#$ is not context free.

Comment

5.48 Solving Language Equations

Background

Task

- A. What language is described by the system $X = aXb + XX + \varepsilon$?
- B. Use Arden's Lemma to solve the system obtained from the minimal automaton for the language "even a 's, even b 's."

Comment

5.49 Multiplicity

Background

Task

Multiplicity

- A. Construct an NFA that accepts a word $x \in \{a, b\}^+$ in exactly $|x|_b$ many ways.
- B. Construct an NFA that accepts a word $x \in 2^+$ in exactly $\text{val}(x)$ many ways (assuming standard binary representation, MSD is first).

Comment

5.50 Small NFAs

Background

Let $L \subseteq \Sigma^*$ be a language. A *fooling set* for L is a list (x_i, y_i) of factored words $i = 1, \dots, n$, such that $x_i y_i \in L$ but for any $i \neq j$ we have $x_i y_j \notin L$ or $x_j y_i \notin L$.

Task

- A. Let L be a regular language with a fooling set of size n . Show that any NFA accepting L must have at least n states.
- B. Let $L_m = \{x \in \{a, b\}^* \mid \#_a x = 0 \pmod{m}\}$. Show that any NFA for L must have at least m states.

Comment

Note that L_m from part (B) is trivially recognizable by a DFA of size m , so in this case nondeterminism does not help.

5.51 Fractional Languages

Background Let $L \subseteq \Sigma^*$ be a regular language and define

$$1/2 L = \{w \in \Sigma^* \mid ww \in L\}.$$

Task

- A. Show that $1/2L$ is again regular.
- B. Derive a bound on $\mu(1/2L)$ in terms of $\mu(L)$ from your construction.
- C. Generalize your construction to $1/n L$ for all positive integers n .
- D. Let $L' = \bigcup_n 1/n L$. Show that L' is regular.

Comment

Chapter 6

Cellular Automata

6.1 Affine Shift Registers

Background

LFSRs have two major advantages: they are lightning fast and there is a good theory to analyze them. Alas, since the underlying functions are linear, there are problems in particular in applications to cryptography (such as stream ciphers). One natural response to such problems is to make the feedback function more complicated (but keep the general feedback-shift mechanism).

Prof. Dr. Wurzelbrunft has suggested a solution for these problems: replace linear feedback functions by affine feedback functions of the form

$$f(\vec{x}) = x_{p_1} + x_{p_2} + \dots + x_{p_r} + 1 \pmod{2}$$

In other words, Wurzelbrunft is suggesting to complement the output bit of the feedback function to define a global map $\overline{F} : \mathbf{2}^k \rightarrow \mathbf{2}^k$. Note that $f(\mathbf{0}) = 1$ so that $\mathbf{0}$ is no longer a fixed point for \overline{F} . The question is: how different is the diagram of \overline{F} from the diagram of F ?

Task

- A. For taps 1, 3, 4, 10, span 10, compare the ordinary version of the FSR to the Wurzelbrunft complemented one: determine all cycles in the diagram of \overline{F} .
- B. As usual, choose the taps in the FSR according to some irreducible polynomial. What is the relationship between the maximum period for the standard linear global map F compared to the Wurzelbrunft map \overline{F} ? Why?
- C. Should Wurzelbrunft get a patent for his idea? Justify your answer.

Comment

A little computational experimentation might sharpen your intuition quite a bit in this case – there will be an obvious claim which turns out to be easy to prove.

6.2 Linear Congruential Generators

Background

LCGs are a standard way to generate mildly random numbers by iteration:

$$x_{n+1} = a x_n + b \pmod{m}$$

The crux of this method is to select the right parameters, otherwise we may encounter catastrophic performance.

Task

- A. Suppose $m = 2^k$ where $k = 32$ or $k = 64$.
- B. What are bad/good choices of a and b in this case?
- C. Now suppose m is a prime, say, a bit smaller than 2^{32} . What are bad/good choices of a and b in this case?
- D. In general, what are the advantages/disadvantages of using prime moduli as opposed to powers of 2?

Comment

A little experimentation may help with this problem, but you might also want to take a look at existing work on the subject. Don't write a book.

6.3 All-Ones

Background

As we have seen in class, the All-Ones problem has a solution over any grid, and in fact any undirected graph. For some simple grids such as narrow strips one can give a direct, geometric solution without having to resort to linear algebra.

Task

- A. Find all solutions of the All-Ones problem for grids of the form $2 \times n$.
 - B. Find all solutions of the All-Ones problem for grids of the form $3 \times n$.
 - C. Suppose the solution for the $n \times m$ grid is not unique. Show that the solution for the $(2n + 1) \times m$ grid cannot be unique either. Hint: look at the sum of two solutions.
-

6.4 Two Simple Cellular Automata

Background

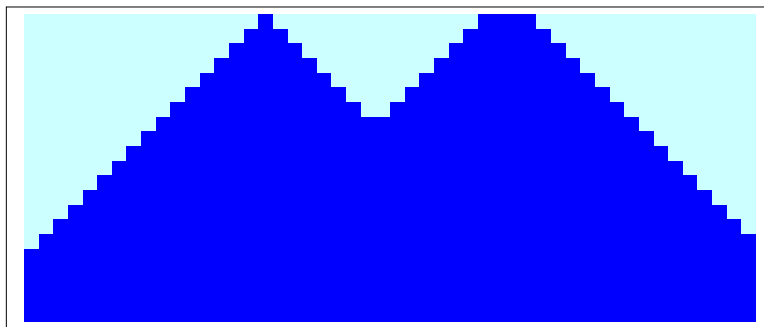
The behavior of a cellular automaton ρ is usually much too complicated to predict what $G_\rho^t(X)$ might look like, given initial configuration X and time $t \geq 0$. But for some special cases a little analysis produces a complete description of $G_\rho^t(X)$ that allows us to predict the state of any particular cell without actually running the whole simulation.

To keep things simple, we assume that we are dealing with cyclic boundary conditions only on a grid of size n (where n is arbitrary).

Task

1. Consider the elementary CA number 254. The local map $\rho : \mathbf{2}^3 \rightarrow \mathbf{2}$ of this CA is equivalent to a simple Boolean function. What is this function?
2. Determine the behavior of the global map G_ρ on all one-point seed configurations. In particular you should compute the transient and period of these configurations.
3. Using the last result, describe the transient and period of an arbitrary configuration. It will be fairly clear what the answer is, try to find a good way to express it.
4. Repeat items (1) through (3) for ECA number 128.

Here is a picture of a the utmost boring evolution of a configuration on ECA 254.



Comment

For the last part (repeat for ECA 128): There is an easy and a hard way to do this. Think before you start writing.

6.5 Additive Cellular Automata

Background

In general the behavior of an elementary cellular automaton ρ on two initial configuration X and Y provides no information about the behavior on $X + Y$ (as usual, $+$ is meant to be addition modulo 2; alternatively you can think about bit-wise xor). But for some special ECA we have additivity:

$$G_\rho(X + Y) = G_\rho(X) + G_\rho(Y)$$

for all configurations X and Y . This is the same effect that we already observed for feedback shift register sequences. Additivity implies that if we can somehow manage to get a nice description for the orbits of one-point seed configurations then we can determine the evolution of any configuration (at least in principle, in actual reality the computations might still be very messy).

Task

1. Show that the local map $\rho : \mathbf{2}^3 \rightarrow \mathbf{2}$ of ECA number 90 is additive.
2. Conclude that the global map $G_\rho : \mathbf{2}^\infty \rightarrow \mathbf{2}^\infty$ of ECA number 90 is also additive.
3. Let X be the one-point seed configuration defined by $X(0) = 1$, $X(x) = 0$ otherwise. Try to find a simple description for $G_\rho^t(X)(x)$. Simple means you should come up with an expression $C(x, t) = \dots x \dots t \dots$ that evaluates to $G_\rho^t(X)(x)$ for all $t \geq 0$ and $x \in \mathbb{Z}$.

Comment

Hint: You will undoubtedly run into binomial coefficients when you do this.

You may find it useful to think about path counting in a rectangular grid (rotated appropriately so things match up properly with the grid of cells). Think about a single pebble in position $x = 0$ at time $t = 0$ that splits and moves one copy to the left, and one to right at time $t + 1$. Two copies in the same spot cancel out.

6.6 Affine Cellular Automata

Background

Suppose $G : \mathcal{C} \rightarrow \mathcal{C}$ is a linear map where $\mathcal{C} = [n] \rightarrow \mathbf{2}$ is vector space over \mathbb{F}_2 . Given any fixed vector $\vec{b} \in \mathcal{C}$ we can define a new map

$$F(\vec{x}) = G(\vec{x}) + \vec{b}$$

Clearly, F reversible if, and only if, G is.

Task

- A. Suppose G is reversible and has order m . Show the the order of F is at most $2m$.
- B. Show that the elementary cellular automata number 60 and number 195 are both reversible on any finite grid with fixed boundary conditions.
- C. Show the hybrid cellular automaton where even-numbered cells use rule 60 and odd-numbered cells use rule 195 is reversible on any finite grid, again with fixed boundary conditions.

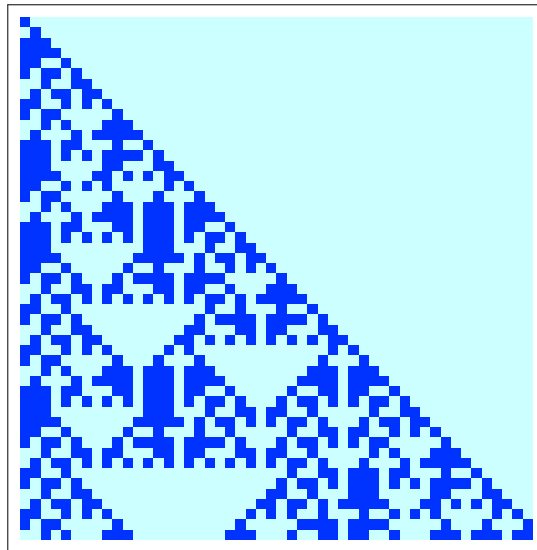
Comment

6.7 Hybrid Cellular Automata

Background

We can generalize the notion of a cellular automaton slightly by allowing different local rules at different cells (i.e., we abandon homogeneity). For this problem, consider in particular automata where the cells alternate between local rule 90 and 150. Thus, locally one cell uses $\rho(x, y, z) = x \oplus z$ but its neighbors use $\rho(x, y, z) = x \oplus y \oplus z$. Moreover, let's agree on fixed boundary conditions. Note that for fixed grid size n there are two choices depending on whether the first cell uses rule 90 or 150.

The picture below shows the evolution of a one-point seed configuration on such a 90-150-hybrid CA.



Task

- Show that the global maps of these automata are linear maps (over \mathbb{F}_2).
- Show that for even n any 90-150-hybrid CA is reversible.
- Determine all reversible 90-150-hybrid CAs.

Comment

For the last part it might be a good idea to implement these automata and check reversibility for some small values of n .

6.8 Additive ECA 90

Background

Recall that the global map of an additive ECA satisfies

$$G_\rho(X + Y) = G_\rho(X) + G_\rho(Y)$$

for all configurations X and Y . For additive cellular automata with cyclic boundary conditions the key is to understand the behavior of “the” one-point seed configuration X_0 since all others can be construed as superpositions of shifts of X_0 . At least in principle, we can then determine the orbits of all configurations given the orbit of X_0 – in actual reality the computations might still be rather messy.

The only a priori upper bound for the transients and periods is 2^n , but it turns out that for additive rules the transients are usually quite short, and the periods are typically also short. Here are the transient/period values for all one-point seed configurations up to size 40 for the additive ECA with rule number 90 and cyclic boundary conditions g (exclusive or of the left and right neighbors). Take some time to study this table, a lot of information is hiding there.

n	t	p	n	t	p
1	–	–	21	1	63
2	–	–	22	1	62
3	1	1	23	1	2047
4	2	1	24	4	8
5	1	3	25	1	1023
6	1	2	26	1	126
7	1	7	27	1	511
8	4	1	28	2	28
9	1	7	29	1	16383
10	1	6	30	1	30
11	1	31	31	1	31
12	2	4	32	16	1
13	1	63	33	1	31
14	1	14	34	1	30
15	1	15	35	1	4095
16	8	1	36	2	28
17	1	15	37	1	87381
18	1	14	38	1	1022
19	1	511	39	1	4095
20	2	12	40	4	24

Here is the distribution of transient/period pairs for $n = 10$.

t/p	#	t/p	#
(0, 1)	1	(1, 1)	3
(0, 3)	15	(1, 3)	45
(0, 6)	240	(1, 6)	720

Needless to say, the tables were obtained by brute force computation.

Task

1. It is easy to check that the local map ρ of ECA number 90 is additive. Verify that the global map G_ρ is also additive.
2. Why are none of these global maps reversible?
3. In the example $n = 10$, the period of any configurations is a divisor of the period of the one-point seed configuration (6 in this case). Is this always the case? Why?
4. Explain the entries in the main table for $n = 2^k$. Describe all the orbits in this special case.

5. How about $n = 2^k \pm 1$ (this is a bit harder).
6. Now consider the bi-infinite grid with the one-point seed configuration X defined by $X(0) = 1$, $X(i) = 0$ otherwise. Find a reasonably simple description for the bit at time $t \geq 0$ in cell $x \in \mathbb{Z}$.

Comment

For the last part, you will undoubtedly run into binomial coefficients. You may find it useful to think about path counting in a rectangular grid (rotated appropriately so things match up properly with the grid of cells). Think about a single pebble in position $x = 0$ at time $t = 0$ that splits and moves one copy to the left, and one to right at time $t + 1$. Two copies in the same spot cancel out.

Extra Credit 1: Do the same for rule 150 (exclusive or of all three cells). This may seem like a minor modification, but things turn out to be considerably more complicated for rule 150.

Extra Credit 2: Explain the rest of the table. One way to tackle this problem is to identify a configuration $(c_0, c_1, \dots, c_{n-1})$ with the polynomial $\sum_{i=0}^{n-1} c_i x^i \in \mathbb{F}_2[x]$. Rule 90 can then be expressed elegantly in terms of the quotient ring $\mathbb{F}_2[x]/(x^n + 1)$. Then use algebra.

6.9 Building Reversible Cellular Automata

Background

For applications in cryptography and also in physical simulations (non-dissipative systems) it is desirable to be able to construct injective cellular automata. There is an old trick due to Fredkin that constructs an injective automaton from a given arbitrary one. Here is the idea, you will have to supply the details. Suppose ρ is the local map of an arbitrary binary cellular automaton. The orbits produced by ρ are of the form

$$X_{n+1} = G_\rho(X_n) = G_\rho^n(X_0).$$

For arbitrary, irreversible ρ there is no way in general to reconstruct X_n from X_{n+1} . But now consider

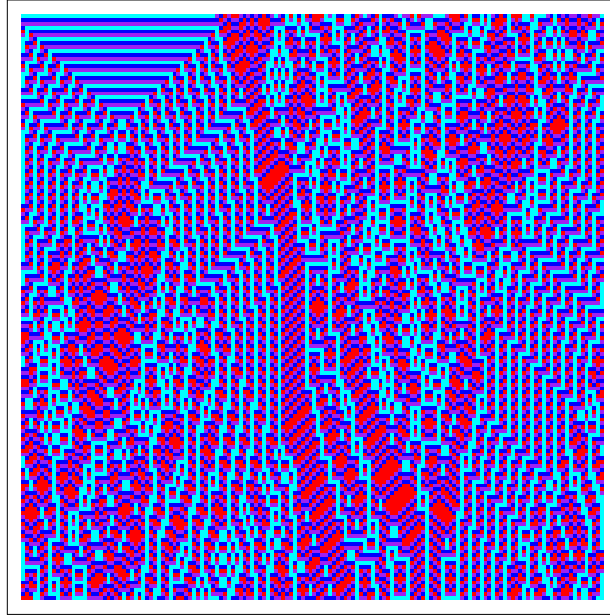
$$X_{n+1} = G_\rho(X_n) \oplus X_{n-1}$$

where \oplus is exclusive 'or' (or addition mod 2), as usual. Then

$$X_{n-1} = G_\rho(X_n) \oplus X_{n+1}$$

and we can go backward – though we need two consecutive configurations to obtain the previous one. This also means we have to have two starting configurations X_0 and X_1 .

So far, all we have is a second-order automaton, but we really want an ordinary injective cellular automaton ρ' based on ρ . To this end, we have to combine two binary configurations into a single configuration over a larger alphabet. Here is the result of this construction applied to ECA 77 (this picture really requires colors, there should be cyan, blue, purple and red).



With a bit of imagination you can almost see how this CA is reversible (there is no loss of information from time t to time $t + 1$), but pictures can be very deceptive.

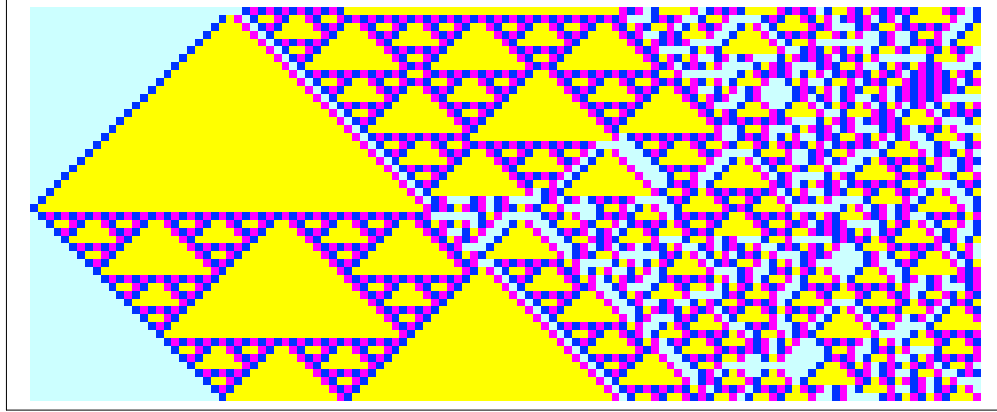
Task

1. Explain exactly how the Fredkin construction of ρ' from ρ works.
2. Prove that your automata ρ' are indeed injective, regardless of the properties of ρ .
3. Let ρ be the CA that is obtained by applying the Fredkin construction to the trivial ECA with number 0. Analyze the behavior of ρ (it's quite boring).

Comment

Needless to say, if you start from an arbitrary ECA ρ and apply Fredkin's trick things become quite interesting. For example, the pseudo-random rule ECA 30 is not reversible, but if we use it to construct a reversible rule we get the following behavior on a one-point seed: first things are fairly regular, but then the patterns become chaotic.

The first 120 steps for $n = 50$.



Incidentally, for $n = 20, 21, 22, 23, 24$ the periods are 85,044; 887,258; 2,217,902; 381,601 and 15,588,247, respectively.

6.10 Counting Boolean Functions

Background

We have seen how to count Boolean functions modulo the equivalence relation induced by inverting some inputs. Several other modifications produce similar classifications of Boolean functions on k inputs.

Task

1. Count the number of orbits when inputs and the output can be inverted.
2. Count the number of orbits when inputs can be rotated. So if the rotation is by one place we have

$$f(x_1, x_2, \dots, x_k) = g(x_2, x_3, \dots, x_n, x_1)$$

3. Count the number of orbits when inputs can be arbitrarily permuted.

$$f(x_1, x_2, \dots, x_k) = g(x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)})$$

4. Calculate the number of classes

Comment

For the last two problems don't try to come up with simple closed-form solutions; they don't exist. But try to write down the answer concisely and elegantly.

6.11 Counting Local Maps

Background

One-dimensional cellular automata are based on local maps $\Sigma^w \rightarrow \Sigma$ where Σ is some alphabet of size k . Depending on the kind of analysis one is interested in, one can sometimes cut back on the number of cases one needs to consider by identifying certain kinds of local maps.

For example, if $\rho(x_1, x_2, \dots, x_w) = \rho'(x_w, \dots, x_1)$ then the two local rules are essentially the same (reflection). Likewise one can ignore rules that are related by a permutation of the alphabet: $\rho(x_1, x_2, \dots, x_w) = \pi^{-1}(\rho'(\pi(x_1), \dots, \pi(x_w)))$. We are just renaming the symbols here, the dynamics are exactly the same.

Task

1. Count the number of local maps of width w modulo reflection.
2. Count the number of local maps of width w modulo cyclic permutations of the alphabet. To simplify matters a little, let's apply the permutation only to the input: $\rho(x_1, x_2, \dots, x_w) = \rho'(\pi(x_1), \dots, \pi(x_w))$ where π is a cyclic permutation of Σ .

Comment

If you feel perky, try part two for arbitrary permutations rather than just cyclic ones.

6.12 Analyzing Simple Elementary Cellular Automata

Background

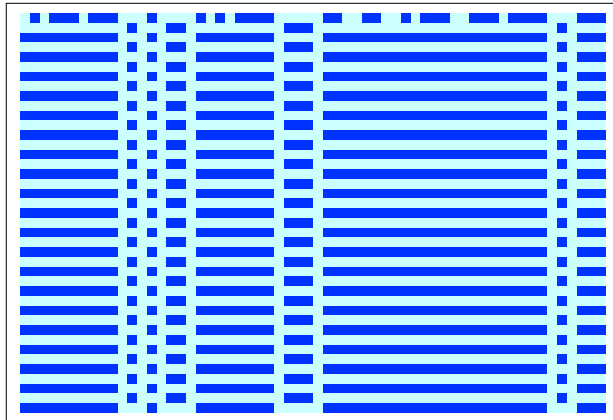
By “analyzing” we mean the following problem: determine the structure of the configuration space \mathcal{C}_n under the action of the global map, for all n . In particular we want a reasonably good description of the possible transient/period pairs (t, p) and a count of the configurations that realize these pairs. Ideally one would want a complete characterization of the configuration X with transient/period (t, p) but that can get very ugly. If you prefer to think about graphs: one would like to understand the weakly and strongly connected components of the functional digraph $\langle \mathcal{C}_n, G_\rho \rangle$.

Of course, analysis is hopeless in general even for ECAs: just think about ECA 30 (pseudo-random) and ECA 110 (universal computation). But for some simple ECAs one can get fairly good answers without too much pain. We will only consider cyclic boundary conditions (though it is tempting, or course, to ask what happens when we switch to fixed boundary conditions).

Task

1. Analyze elementary CA number 1.
2. Analyze elementary CA number 128.
3. Analyze elementary CA number 170.
4. Analyze elementary CA number 254.

Here is a fairly typical picture of a the evolution of a random seed configuration on ECA 1.



Comment

Try to come up with elegant answers, but don't expect miracles.

Chapter 7

Algebra

7.1 Modular Multiplication

Background

Given a modulus m , the numbers coprime with m form a group under multiplication modulo m , the so-called *multiplicative subgroup*. This group \mathbb{Z}_m^* appears often in number theory and combinatorics; the map $\Phi(m) = |\mathbb{Z}_m^*|$ is known as *Euler's totient* function.

While \mathbb{Z}_m^* is the most important group of modular numbers under multiplication there are others. For example, $\{3, 9, 21, 27\}$ forms a group for $m = 30$ with 21 as the unit. We will call any group $G \subseteq \{1, 2, \dots, m-1\}$ with modular multiplication as operation a *strange multiplicative group*. The key to identifying these strange groups is a simple observation: the unit element e of G must be idempotent: $e^2 = e \pmod{m}$. To identify the idempotents modulo m it is convenient to consider divisors a of m such that a and m/a are coprime. We will call these *prompt* divisors. Recall that by the Chinese Remainder Theorem \mathbb{Z}_m is isomorphic to $\mathbb{Z}_a \times \mathbb{Z}_{m/a}$ for any prompt divisor a .

Task

- A. Verify that $\{3, 9, 21, 27\}$ forms a group for $m = 30$.
- B. Give a characterization of idempotents modulo m in terms of the prime factorization of m . Use this to count the number of idempotents.
- C. Show that every idempotent e is the unit of a group $G_e = e\mathbb{Z}_m^*$.
- D. Show that all the G_e are disjoint.
- E. Show that for every prompt divisor a of m we have $a\mathbb{Z}_m^* = \{x \in \mathbb{Z}_m \mid \gcd(x, m) = a\}$.
- F. Use the last result to give a simpler description of the groups G_e .
- G. Determine the order of the groups G_e .
- H. Show that every strange group is a subgroup of some G_e .

Comment

Fermat's little theorem and Euler's totient function are helpful for part (B). One can show that every strange group is a subgroup of some G_e but that requires quite a bit more work.

You might find it useful to do a little experimentation with, say, $m = 100$.

7.2 An Associative Operation

Background

Many operations in algebra are clearly associative. However, in some cases associativity is far from obvious and requires a proof.

For this problem, write $[x]$ for the integer nearest to $x \in \mathbb{R}$. More precisely, let $[x] = \lfloor x + 1/2 \rfloor$ so that $[3.5] = 4$. Consider the map $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$\pi(x, y) = \begin{cases} x + y^2 & \text{if } x \neq \max(x, y), \\ x(x + 1) + y & \text{otherwise.} \end{cases}$$

The last map is a bijection between \mathbb{N} and the first quadrant in the plane, see part (A). Here is a slightly different map: $\rho : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{Z}$

$$\rho(x) = ([\sqrt{x}], x - [\sqrt{x}]^2)$$

Let $W \subseteq \mathbb{N} \times \mathbb{Z}$ be the range of ρ .

Task

- Show that π is a bijection and determine the unpairing functions.
- Determine W , the range of ρ and show that ρ is a bijection between \mathbb{N} and W .
- Show that the binary operation on \mathbb{N} defined by

$$a * b = a + b + 2[\sqrt{a}][\sqrt{b}]$$

is associative.

Comment

Part (A) is just a warm-up, but part (B) is useful for (C). The latter can also be handled by brute force, but that's a mess.

7.3 Generating Permutations and Functions

Background

We have seen that all permutations of $[n]$ can be obtained from the cyclic shift $(2, 3, \dots, n, 1)$ and the transposition $(2, 1, 3, \dots, n)$ by composition. Needless to say, there are many other generating sets (e.g., the set of all transpositions) and there are quite a few of size 2.

Also note that by adding just one more function (not a permutation) one can obtain all functions $[n] \rightarrow [n]$ by composition.

Task

- A. Characterize all possible generating sets of size 2 for the group of all permutations of $[n]$.
- B. Characterize all possible generating sets of size 3 for the monoid of all functions $[n] \rightarrow [n]$.
- C. Find a small set of generators for the alternating group on n points. You might want to find an analogue to the fact that the full symmetric group is generated by transpositions first.

Comment

As always, “characterize” means that you have to find some condition that holds if, and only if, f and g are generators for the symmetric group. Try to find a simple, elegant condition. Then prove that your condition really works. You can use the fact that the two permutations above generate the symmetric group, but you have to prove everything else.

Note that the corresponding problem for general relations (under relational composition) is much more difficult. There is no fixed size set of generators, but the proof is hard.

7.4 Matrices and Words

Background

The collection of all words over the alphabet $\{a, b\}$ can be construed as a monoid: the associative operation is concatenation and the neutral element is ε , the empty word. This monoid, usually written $\{a, b\}^*$, is well-known to be *free*: given an element $x \in \{a, b\}^*$ there is exactly one way to write it as a product of a 's and b 's.

Here is another, more algebraic way to generate the free monoid over two elements. Consider the two integer matrices

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Let $M \subseteq \mathbb{N}^{2 \times 2}$ be the monoid generated by A and B .

Task

- A. Show that any matrix in M can be written in precisely one way as a product of A and B .
- B. Turn the last result into a decomposition algorithm and discuss its running time.
- C. Explain how to represent the free monoid over $k > 2$ generators in terms of matrices.
- D. Show that a 2×2 matrix over \mathbb{N} belongs to M if, and only if, M had determinant 1.

Comment

In a similar way we can produce the free group with two generators by including the inverse matrices of A and B .

7.5 Boolean Rings

Background

A *Boolean ring* is a ring (with unity, as always) where every element is idempotent: $x^2 = x$. A classical example for a Boolean ring is \mathbb{Z}_2 .

Task

- A. Show that every Boolean ring has characteristic 2.
- B. Show that every Boolean ring is commutative.
- C. Suppose $\langle B, \vee, \wedge, \neg, \text{tt}, \text{ff} \rangle$ is a Boolean algebra. Define operations plus $+$ and times $*$ by

$$\begin{aligned}x + y &= (x \vee y) \wedge \neg(x \wedge y) \\x * y &= x \wedge y\end{aligned}$$

and set $0 = \text{ff}$ and $1 = \text{tt}$. Show that this produces a Boolean ring on B .

- D. Explain how to obtain a Boolean algebra from a Boolean ring.

Comment

7.6 Boolean Algebras

Background

The truth values of propositional logic have a (fairly simple) algebraic structure under the operations \vee , \wedge and \neg . This particular 2-element structure generalizes naturally to the class of Boolean algebras. Boolean algebras can be axiomatized as algebraic structures of signature $(2, 2, 1; 0, 0)$ as follows.

$$\begin{aligned}x + (y + z) &= (x + y) + z & x \cdot (y \cdot z) &= (x \cdot y) \cdot z \\x + y &= y + x & x \cdot y &= y \cdot x \\x + 0 &= x & x \cdot 1 &= x \\x + (y \cdot z) &= (x + y) \cdot (x + z) & x \cdot (y + z) &= (x \cdot y) + (x \cdot z) \\x + \bar{x} &= 1 & x \cdot \bar{x} &= 0\end{aligned}$$

For any term t in a Boolean algebra define its dual t^{op} to be obtained from t by interchanging $+$ and \cdot , as well as 0 and 1 .

Task

- A. Establish the following Duality Principle: An equation $s = t$ is valid if, and only if, its dual $s^{\text{op}} = t^{\text{op}}$ is valid.

B. Show that the following equalities concerning addition and multiplication hold in all Boolean algebras.

$$\begin{array}{ll} x + x = x & x \cdot x = x \\ x + x \cdot y = x & x \cdot (x + y) = x \\ x + 1 = 1 & x \cdot 0 = 0 \end{array}$$

C. Show that the following equalities concerning complementation hold in all Boolean algebras.

$$\overline{x + y} = \bar{x} \cdot \bar{y} \quad \overline{x \cdot y} = \bar{x} + \bar{y} \quad \overline{\bar{x}} = x$$

Comment Make sure to come up with an elegant, coherent proof for the duality part.

7.7 Boolean Algebras without Times

Background

Traditionally, Boolean algebras are axiomatized using two binary operations plus $+$ and times \cdot , a unary operation complementation $\bar{}$, and constants 0 and 1. Here is one possible set of axioms.

$$\begin{array}{ll} x + (y + z) = (x + y) + z & x \cdot (y \cdot z) = (x \cdot y) \cdot z \\ x + y = y + x & x \cdot y = y \cdot x \\ x + 0 = x & x \cdot 1 = x \\ x + \bar{x} = 1 & x \cdot \bar{x} = 0 \\ x + (y \cdot z) = (x + y) \cdot (x + z) & x \cdot (y + z) = (x \cdot y) + (x \cdot z) \end{array}$$

Now remove all the axioms involving times (right column, and last row left column), call the resulting system (A). In (A) we can still introduce times as a definition:

$$x \cdot y := \overline{\bar{x} + \bar{y}}$$

Task

- Show that all the missing axioms can be derived from (A) using this definition of times.
- How would you cope with a system where all the axioms involving complementation are missing?

Comment

7.8 A Transformation Semigroup

Background

It is easy to construct the minimal DFA M for the language

$$L = \{(aab)^i \mid i \geq 0\}$$

over alphabet $\{a, b\}$ by hand. M has 4 states, so let's agree that state 4 is the sink. Let $\mathcal{T} \subseteq \text{Fct}(Q, Q)$ be the transformation semigroup of this automaton.

Task

- Construct \mathcal{T} from its generators δ_a and δ_b . Make sure to keep track of the witnesses w such that $\delta_w \in \mathcal{T}$ as produced by the old spanning tree algorithm, they will be useful for part D.
- Explain why \mathcal{T} contains a null. How is this null element related to L ?
- Explain why \mathcal{T} does not contain a one element.
- What are the simplification rules that describe the semigroup in terms of words over $\{a, b\}$? For simplicity, write 0 for the null element.

Comment

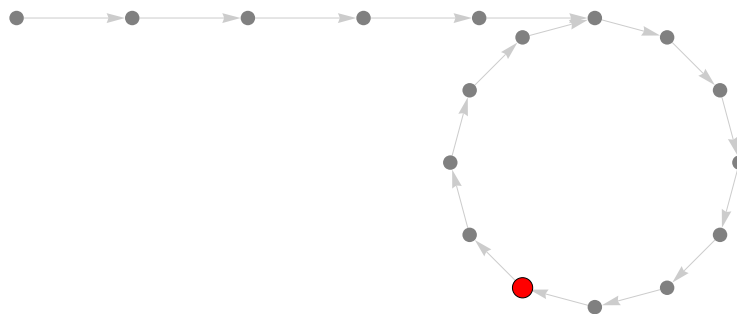
\mathcal{T} has 11 elements, make sure you get them all. 5 simplification rules suffice for \mathcal{T} ; you can look at the language to find out what they are, or you can extract them from the generating algorithm. E.g., you will want to have a rule $bb \rightarrow 0$.

7.9 Floyd goes Algebraic

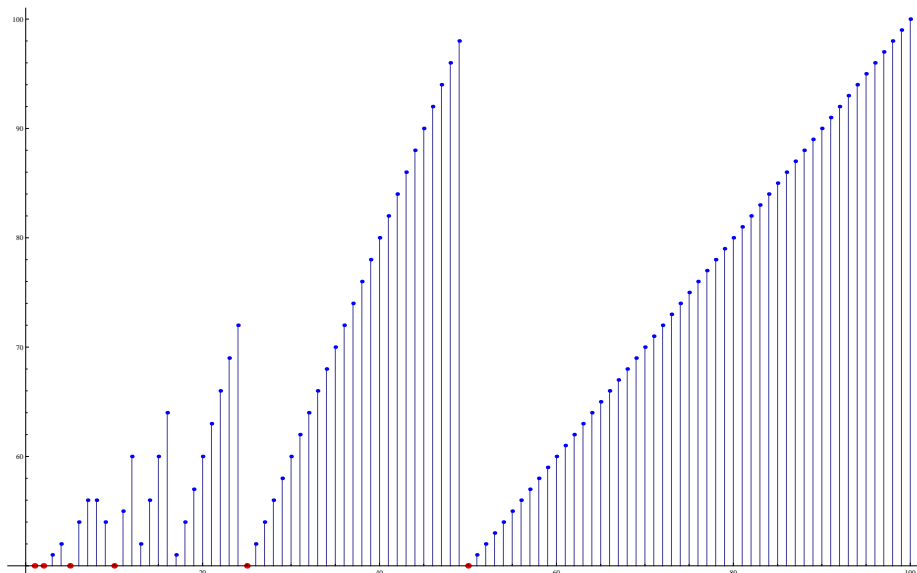
Background

One of the most elementary results about finite semigroups is that every element in the semigroup has an idempotent power. Recall that x is idempotent if $x^2 = x$. Then the claim is that for S any finite semigroup and $a \in S$, there exists some integer $r \geq 1$ such that a^r is idempotent.

Ignoring algebraic aspects for a moment, note that the powers of a (the points a^i , $i \geq 1$) must form a lasso since S is finite. Hence we can associate a with a transient $t = t(a) \geq 0$ and a period $p = p(a) \geq 1$.



We can think of the powers of a as the orbit of a under the map $x \mapsto ax$, so we can apply Floyd's trick to find a point on the loop (the big, red dot above). Let's call the time when the algorithm finds this point the *Floyd time* $\tau(t, p)$, an integer in the range 0 to $t + p - 1$. The next picture shows the Floyd times for $t = 50$ and $p = 1, \dots, 100$. The red dots indicate the values of p where the Floyd time is $t = 50$.



Inquisitive minds will wonder if there is any connection between the Floyd time and the idempotent from above.

Task

- Prove the claim about idempotents in finite semigroups. Hint: think about the Floyd time.
- Show that there is exactly one idempotent among the powers of a .
- Give a simple description of the Floyd time $\tau(t, p)$ in terms of the transient t and period p .
- Explain the plot of the Floyd times above.
- Show that the powers of a that lie on the loop form a group with the idempotent as identity.

Comment

For extra credit you might (re-)consider the question of what happens in Floyd's algorithm when one changes the velocities of the particles to $1 \leq u < v$.

7.10 Polynomial Equations Mod 2

Background

We have seen that Diophantine equations are hard: it is undecidable whether a polynomial with integer coefficients has an integer solution. By contrast, modular arithmetic is easy in the sense that one can conduct

brute force search over the finitely many possible values. Arithmetic over \mathbb{Z}_p , p a prime, is particularly interesting since we are dealing with a field in this case.

Task

1. Show that any function $f : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_2$ can be expressed in terms of a polynomial $P(x_1, \dots, x_n)$.
2. Show that one can solve a single polynomial equation $P(x_1, \dots, x_n) = 0$ over \mathbb{Z}_2 in polynomial time.

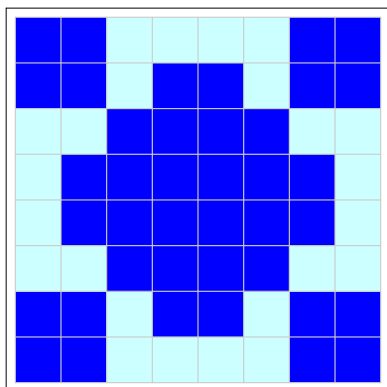
7.11 Chessboards and Lights

Background

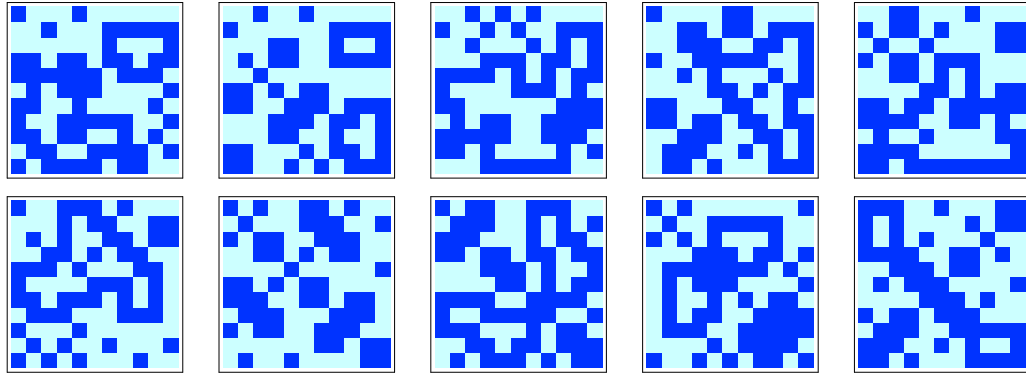
Suppose a chessboard is equipped with light bulbs, one bulb for each square. Pushing a square acts like a toggle, it switches the light bulb from on to off and from off to on. Initially, all lights are off. Our goal is to turn all the lights on. The catch is that the wiring is a bit faulty so that pushing a square toggles not just the light of that particular square but also of the (at most four) neighboring squares. Of course, it's not at all clear that this puzzle has a solution. With a little effort one can prove the following theorem:

Theorem: The lights-on puzzle has a solution for any n by n chessboard.

For a 4 by 4 board this can be verified by a little experimentation but an 8 by 8 board is hard to deal with. Here is a solution (which turns out to be unique if we disregard the order in which the squares are pushed):



For the 11 by 11 grid there are 64 solutions. If we count modulo rotations and reflections (the natural action of D_4 on the board), only 10 patterns remain (why 10?):



Given the existence theorem, one could perform a brute force search for a solution. Alas, there are 2^{n^2} possible configurations, on an n by n board, so a more clever line of approach is needed. feasible.

Task

1. Find a fast algorithm to compute the number of solutions for the lights-on puzzle on an n by n grid.
2. Find a fast algorithm to compute an actual solution for the lights-on puzzle on an n by n grid.

Comment

Fast definitely means not exponential in n . Think vector spaces over the two-element field.

Extra Credit 1: Find a solution for the 40 by 40 grid (plot a picture, it's easy to check visually whether your solution is correct).

Extra Credit 2: Prove the existence theorem.

7.12 Shrinking Dimension

Background

As we have seen in class, there is a unique finite field \mathbb{F} of size p^k for any prime p and $k \geq 1$. In one standard implementation we then think of \mathbb{F} as a vector space of dimension k over \mathbb{F}_p , so the field elements are vectors of modular numbers. However, it is sometimes more convenient to deal with a lower-dimensional vector space over a larger ground field. More precisely, we may have a chain of subfields

$$\mathbb{F}_p \subseteq \mathbb{K} \subseteq \mathbb{F}$$

and we can consider \mathbb{F} as a vector space over \mathbb{K} . Alas, this only works under special circumstances which will be described in this problem.

Fix some prime characteristic p throughout.

Task

- A. Show that the following are equivalent, $1 \leq \ell \leq k$:

- (a) ℓ divides k
- (b) $p^\ell - 1$ divides $p^k - 1$
- (c) $x^\ell - 1$ divides $x^k - 1$ (in the polynomial ring $\mathbb{F}_p[x]$).

B. Show that if \mathbb{K} is a subfield of \mathbb{F} then $\mathbb{K} = \mathbb{F}_{p^\ell}$ where ℓ divides k .

C. Show that if ℓ divides k then \mathbb{F}_{p^ℓ} is a subfield of \mathbb{F} .

Comment

The last item is the hardest; think splitting fields.

7.13 Building A Finite Field

Background

As we have seen in class, there is a unique finite field of size p^k for any prime p and $k \geq 1$. Needless to say, the case $p = 2$ it is particularly interesting for actual implementations: the prime field consists just of bits 0 and 1 with *xor* as addition and *and* as multiplication.

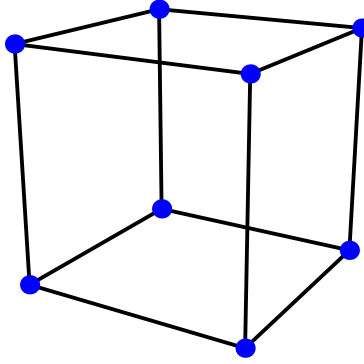
Task

- A. Show how to construct the finite field \mathbb{F} of size 256. What data structures would you use, how would you implement arithmetic in this field.
 - B. How many primitive elements are there in this field?
 - C. What are all the subfields of \mathbb{F} ? Why?
 - D. If we had constructed a field of size 32, what would the subfields be?
 - E. What is the main difficulty in doing a similar construction for the field of size 2^{1024} ?
-

7.14 Moving Cubes

Background

Dihedral groups describe the motions of regular polygons in the plane (rigid motions corresponding to rotations plus reflections). Needless to say, motions also make sense in three dimensions – and things become a bit more complicated because of the additional degrees of freedom.



In this problem, consider the group G of all the rigid motions that move a cube in 3-space back to itself. These are all rotations, we will not consider reflections here. We can think of G as a subgroup of \mathbb{S}_8 , the full symmetry group on 8 points.

Task

- A. Produce a table of all group elements together with their orders.
- B. Find a (small) set of generators for G .
- C. Determine the cycle decomposition of all the elements of G .

Extra Credit: What happens if one allows reflections?

Comment

In order to get a handle on the group elements, first think about possible axes of rotation. For example, an axis could go through the centers of two opposite faces. The critical classification is: face-centered, edge-centered, vertex-centered. Unless you have perfect geometric intuition it might be a good idea to use a physical model of a cube to find the rotations.

7.15 Characteristic 2

Background

As we have seen in class, there is a unique finite field of size p^k for any prime p and $k \geq 1$. When $p = 2$ it is particularly easy to give efficient implementations of the arithmetic in these fields.

Task

- A. Explain how to implement the finite field \mathbb{F}_{2^5} .
- B. How difficult is it to implement addition and multiplication in your system? How about division?
- C. How many primitive elements are there in \mathbb{F}_{2^5} ?
- D. What are all the subfields of \mathbb{F}_{2^5} ? Why?

E. How does all of this carry over to \mathbb{F}_{2^k} ? What is the main difficulty in implementing the field of size 2^{1000} ?

7.16 Redundant Field Representations

Background

As we have seen, it may be computationally advantageous to implement a finite field \mathbb{F}_{2^k} by embedding it in a quotient ring $\mathbb{K} = \mathbb{F}_2[x]/(x^n + 1)$. For example, \mathbb{F}_4 can be embedded into $\mathbb{F}_2[x]/(x^3 + 1)$ (a generator for the multiplicative group is $x + 1$).

For this method to be of any practical use the least n such that that $\mathbb{F}_2[x]/(x^n + 1)$ embeds \mathbb{F}_{2^k} cannot be much larger than k . Let's write $\eta(k)$ for this "embedding number". For example, it turns out that $\eta(8) = 17$. The following lemma can be found in an otherwise excellent paper from 1998.

Lemma: $\eta(k) = \min(n > k \mid k = \text{order of } 2 \text{ in } \mathbb{Z}_n^*)$

Note that \mathbb{F}_{2^k} is a subfield of \mathbb{F}_{2^ℓ} if, and only if, k divides ℓ .

Task

- A. Compute $\eta(5)$ and $\eta(10)$ according to this lemma.
- B. Conclude that the lemma is false.

Comment

The correct condition is that $x^n + 1$ has an irreducible factor of degree d such that $k \mid d$. For example, $x^{17} + 1 = (x + 1)(x^8 + x^5 + x^4 + x^3 + 1)(x^8 + x^7 + x^6 + x^4 + x^2 + x + 1)$ and this is the first time an irreducible factor of degree 8 occurs (whence $\eta(8) = 17$).