AICC 1 - Notes and Summary

Faustine Flicoteaux

Fall Semester 2024

Contents

Introduction

These are my notes for the Advanced Information, Communication and Computation I (CS-101) course given during the fall semester of 2024 at EPFL. Please note that the content is not mine but belongs to Professor Thomas Bourgeat and Professor Tanja Käser, who taught it. I have however changed some formulations, added definitions from other sources and personal notes, when I thought it useful.

This summary is not exempt of errors. If you find one, you can contact me at my EPFL e-mail address: `faustine.flicoteaux@epfl.ch` or through the GitHub page `https://github.com/FocusedFaust/LectureNotes`.

Part I

Logic and Mathematical Reasoning

Chapter 1

Propositional Logic

1.1 What is Logic?

Logic is the "language of mathematics". It is more precise than human language by avoiding expression interpretation (imprecise 'if', 'or', 'then', ...), which you will see when presented with propositions later. Logic is also the basis for mathematical proofs and automated reasoning, which we will both study during the semester. Finally, logic is omnipresent in computing (if condition is true, then do something).

Logic is about statements that are either **true** or **false**.

We will start by studying *propositional logic*, which is the most basic form of logic.

First, a bit of background Logic was first developed by Greek philosophers to formalize reasoning. Then, modern mathematicians formulated propositional logic.

Though basic, propositional logic introduces many fundamental concepts for mathematics (and computer science) such as formal language, variables and operators, axioms, inference, proof, truth value, ...

A point of importance is that, for anything expressed in propositional logic, we can automatically decide whether it is true or false, which is not the case for other logics.

1.2 Propositions

A *proposition* is a declarative sentence that is either true or false.

Atomic Propositions An atomic proposition is a proposition that cannot be expressed as simpler propositions. We use letters to denote these propositional variables: $p, q, r, s, ...$
A proposition that is always true is denoted by T
A proposition that is always false is denoted by F

Compound Proposition Compound propositions are constructed using logical connectives and other propositions. The logical connectives are the following, ordered by precedence:

1. Negation $\neg$
2. Conjunction $\wedge$
3. Disjunction $\vee$
4. Implication $\Rightarrow$
5. Biconditional $\Leftrightarrow$

Truth tables A *truth table* lists all possible truth values of the propositional variables occurring in a compound proposition and the corresponding truth values of the compound proposition[1].

1.2.1 Logical connectives

Negation Let p be a proposition. The negation of p, denoted by $\neg p$ (or $\bar{p}$), is the statement "It is not the case that p".

The proposition $\neg p$ is read "not p". The truth value of $\neg p$ is the opposite of the truth value of p.

p	¬ p
T	F
F	T

Conjunction Let p and q be propositions . The *conjunction* of p and q, noted $p \wedge q$, is the proposition "p and q". The conjunction is true when *both* p and q are true and is false otherwise.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction Let p and q be propositions. The disjunction of p and q, noted $p\vee$ is the proposition "p or q". It is false when both p and q are false and is true otherwise.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

In natural language, "or" has two distinct meanings : inclusive or exclusive. Inclusive means that one or both of the propositions can be true ("I am reading this document and I am in class"). Exclusive means that both cannot be true at the same time ("I am listening to the teacher or reading my notes").

[1] There are many websites useful for generating truth table (which is rather tiresome to do by hand). I personally recommend `https://truth-table.com/`.

Exclusive Or Exclusive or, noted $p \oplus q$, is also named "xor". It is true when either p or q is true but not both.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Implication Let p and q be propositions. The conditional statement $p \Rightarrow q$ is the proposition "if p then q". It is false when p is true and q is false and is true otherwise.

In the conditional statement $p \Rightarrow q$, p is called the hypothesis (or antecedent or premise) and q is called the conclusion (or consequence).

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

If p is false, the implication is always true. If q is true, the implication is also always true.

$p \Rightarrow q$ is different from $q \Rightarrow p$: "If it is sunny, I will go to the plage du pélican" is not equal to "If I go to the plage du pélican, it will be sunny" (I wish it were true).

One way to view the logical conditional is to think of an obligation or contract. A politician says "If I am elected, then I will lower the taxes." If the politician is elected but the taxes are not lowered, then we can say that they broke the campaign pledge. However, if the politician is not elected, no one will care.

The **inverse** is the proposition $\neg p \Rightarrow \neg q$.

The **converse** is the proposition $q \Rightarrow p$.

The **contraposive** is the proposition $\neg q \Rightarrow \neg p$
Converse and inverse are logically equivalent.

Biconditional Let p and q be propositions. The conditional statement $p \Leftrightarrow q$ is the proposition "p if and only if q". It is true when both p and q have the same truth value.

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

1.2.2 Classification of compound propositions

Tautology A *tautology* is a proposition that is always true.

For example, $p \vee \neg p$, "*When I get home, I will either read a book or do any other thing*", "*If we do not succeed, we run the risk of failure* (Dan Quayle)".

Contradiction On the other side, a *contradiction* is a proposition that is always false.

For example, $p \wedge \neg p$, "*It is not blue and red, it is red and blue*" and "*Never say never*" are never (☺) true.

Contingency A *contingency* is a proposition that is neither a tautology nor a contradiction. It can be either true or false.

For example, we have the simplest of all: p.

1.2.3 Propositional satisfiability

A compound proposition is *satisfiable* if there is an assignment of truth values to its variables that make it true. This means that the proposition can be true, and thus that it is either a tautology or a contingency.

When no such assignment exists, the proposition is *unsatisfiable*. Therefore, it is a contradiction.

Modelling a problem as a compound proposition and evaluating its satisfiability is the equivalent to asking "*Is there a solution?*".

1.2.4 Logic equivalences

Two compound propositions p and q are *logically equivalent* if $p \Leftrightarrow q$ is a tautology. We write this as $p \equiv q$. This means that the truth tables output the same values for the same variable truth assignment.

1.2.5 Normal forms

It is possible to convert an arbitrary proposition into its canonical form, also called **normal form**. This is useful to prove theorems, because two propositions are equivalent if their normal forms are equivalent themselves.

Disjunctive Normal Form The DNF is the *disjunction* of one or more *conjunctions* of one or more variables, called the **minterms**. The full DNF is the DNF where every variable or its negation is represented exactly once in every minterm.

To construct the DNF, we use the rows of the truth table where the proposition is **true** to construct minterms: if the variable is true in that row, use it directly. If it is false, use its negation. We then connect the minterms with $\vee$s.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

For example, the proposition $p \oplus q$ gives the following truth table. We use the second and third rows to get two minterms: $(p \wedge \neg q)$ and $(\neg p \wedge q)$. The (full) DNF is then $(p \wedge \neg q) \vee (\neg p \wedge q)$.

Conjunctive Normal Form The CNF is a *conjunction* of one or more *disjunctions* of one or more variables (a clause or **maxterm**). The full CNF is the CNF where every variable or its negation is represented exactly once in every minterm.

To construct the CNF, we use the rows of the truth table where the proposition is **false** to construct maxterms: if the variable is true in that row, use its negation. If it is false, use it directly. We then connect the clauses with $\wedge$s.

For our earlier example, we would use the first and last rows of the truth table to get two clauses: $(\neg p \vee \neg q)$ and $(p \vee q)$. The (full) CNF becomes $(\neg p \vee \neg q) \wedge (p \vee q)$.

Finding a CNF/DNF without a truth table We produce a series of equivalences, stating with the proposition.

First, we **eliminate implications**: $p \Rightarrow q$ becomes $\neg p \vee q$.

Secondly, we move **negations inward** with DeMorgan's law.

Finally, we use the **distributive** and **associative** laws.

This leaves us with a CNF or DNF. This method is faster than the truth table one, especially on long propositions, but it does not guarantee a full CNF or DNF.

Chapter 2

Predicate Logic

When propositional logic is not enough, we use predicate logic to talk about objects, their properties and relations.

The heart of predicate logic is statements involving *variables*. The truth value of $P(x)$ depends on the concrete value of x, such as $Q(x, y) := x + y = 5$. Is $Q(2, 4)$ true?

Connectives from propositional logic can be applied to predicate statements. We can also construct expressions from predicates and logic connectives containing variables ($R(x, y) := P(x) \Rightarrow P(y)$).

2.1 Quantifiers

The domain of a proposition is all the possible values of x (for example: integers, animals, colors). Quantifiers are used to express to which extent a propositional function is true over all values of the domain U of its variables.

Universal quantifier The universal quantifier is the statement "$P(x)$ *is true for all values of* x *from its domain*". We usually write that "$\forall x, P(x)$" and it is read as "*for all* x*,* $P(x)$ *is true*".

To show that $\forall x P(x)$ is false, we have to find a single x for which $P(x)$ is false. That x is a *counterexample*.

Existential quantification The existential quantification is the statement "*there exists an element* x *from domain* U *such that* $P(x)$ *is true*". We usually write that "$\exists x, P(x)$".

To show that $\exists x P(x)$ is true, we have to find a single x for which $P(x)$ is true. That x is a *witness*.

Uniqueness quantification The uniqueness quantification is the statement "*there exists a unique element* x *from domain* U *such that* $P(x)$ *is true*". We usually write that "$\exists! x, P(x)$".

2.2 Finite domains

If the domain U is finite, we can list all elements of $U = \{x_1, x_2, ..., x_{n\in\mathbb{R}}\}$ (although it can be long with countable infinite) and we can express quantified statements using propositional logic.

$\forall x P(x)$ is the same as $P(x_1) \wedge P(x_2) \wedge ... \wedge P(x_n)$.
$\exists x P(x)$ is the same as $P(x_1) \vee P(x_2) \vee ... \vee P(x_n)$.

2.3 Precedence and Binding

The quantifiers $\forall$, $\exists$ and $\exists!$ have precedence over the all logical connectives.
A quantifier **binds** the variable of a propositional function :

- $P(x)$ is a proposition with a **free variable** x.
- $\forall x P(x)$ is a proposition with a **bound variable** x.

2.4 Validity and Satisfiability

Validity A statement involving predicates and quantifiers with all variables bound is **valid** if it is true for all interpretations, meaning that it is true regardless of what the constants and variables mean.

$\forall x(P(x) \Rightarrow (P(x) \vee Q(x)))$ is a tautology. It is therefore valid (and satisfiable).

Satisfiability That same statement is **satisfiable** is it can be true, meaning that there is some interpretation in which it is true.

For example, consider the proposition $\forall x P(x, x)$. If $P(x, y)$ is $x \geq w$, then the proposition is true. However, if $P(x, y)$ is $x \neq y$, the proposition is false.

Unsatisfiability That same statement is unsatisfiable if it is a contradiction if there is no interpretation for which it is true.

Let us consider the statement $\forall x(P(x) \wedge \neg P(x))$. It can be assimilated to the contradiction $p \wedge \neg p$. Therefore, it is unsatisfiable.

2.5 Truth values of Quantifiers

- Method 1: perform an *equivalence proof* to a true or false statement.
- Method 2: reason about the values of the domain
  - To show a state is True, show that its *negation* is False.
  - To show that a universally quantified statement is True, analyse different sub-cases of the domain.
  - Check carefully what interesting values may occur in the domains.
  - Find *witnesses* for existential quantification.
  - Find *counterexample* for universal quantification.

Quantifiers truth values

Part II

Basic Structures

Chapter 3

Binary relations

Definition A binary relation R from a set A to a set B is a subset $\mathrm{R} \subseteq A \times B$.

Example Let $A = 0, 1$ and $B = a, b, c$, then

- $A \times B = (0, a), (0, b), (0, c), (1, a), (1, b), (1, c)$
- $R_1 = (0, a), (0, b), (1, a)$is a relation from A to B
- $R_2 = (0, a), (1, b)$ is a relation from A to B

Functions and Relations

A function $f : A \to B$ can also be defined as a subset of $A \times B$, meaning, as a relation.

A function f from A to B contains one, and only one ordered pair (a, b) for every element $a \in A$.

3.1 Symmetric and Antisymmetric Relations

Definition of symmetry

A relation R on a set A is called *symmetric* if, and only if, $(b, a) \in R$ whenever $(a, b) \in R$, for all $a, b \in A$.

Therefore, R is symmetric if, and only if, $\forall x \forall y((x, y) \in R \to (y, x) \in R)$.

Definition of antisymmetry

A relation R on a set A is called *antisymmetric* if, and only if, $(b, a) \in R$ and $(a, b) \in R$ then $a = b$, for all $a, b \in A$.

Therefore, R is antisymmetric if, and only if, $\forall x \forall y((x, y) \in R \wedge (y, x) \in R) \to x = y$.

Remark Symmetric and antisymmetric are not opposites of each other

Personal remark There is only one relation for any set A that is both symmetric and antisymmetric. It is the relation $R = \{(a, a) | a \in A \ \forall a\}$

3.2 Transitive Relations

Definition

A relation R on a set A is called *transitive*, if and only if, whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for all $a, b, c \in A$.

In other words, R is transitive if and only if, $\forall a \forall b \forall c((a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$.

3.3 Number of Relations on a Set

- $A \times A$ has $|A|^2$ elements when A has $|A|$ elements.
- Every subset of $A \times A$ can be a relation.
- Therefore, there are $2^{|A|^2}$ relations on a set A.

3.4 Combining Relations

Given two relations R_1 and R_2, we can combine them using basic set operations to form new relations, namely

- $R_1 \cup R_2$
- $R_1 \cap R_2$
- $R_1 - R_2$
- $R_2 - R_1$

3.5 Composition of Relations

Definition

Let R be a relation from a set A to a set B. Let S be a relation from B to a set C ($R : A \rightarrow B$ and $S : B \rightarrow C$)

The composite of R and S is the relation consisting of ordered pairs (a, c), where $a \in A$, $c \in C$, and for which there exists an element $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$.

Personal Remark In other words, the composite is the relation mapping elements of A to elements of C according to the relations R and S.

We denote the composite of R and S by $S \circ R$.

3.6 Equivalence Relations and Classes

Definition

A relation on a set A is called an equivalence relation if, and only if, it is reflexive, symmetric and transitive.

Two elements a and b that are related by an equivalence relation are called equivalent. The notation $a \sim b$ is often used to denote equivalent elements.

Example The relation $R = \{(a,a),(a,b),(b,b),(b,a),(c,c)\}$ on the set $A = \{a,b,c\}$ is an equivalence relation.

The relation $R = \{(a,b) \in \mathbb{R} \times \mathbb{R} \mid a - b \in \mathbb{Z}\}$ is an equivalence relation on the set $\mathbb{R}$.

Definition

Let R be an equivalence relation on a set A. The set of all elements that are related to an element a of A, in that relation R, is called the equivalence class of a.

We denote the equivalence class of an element a $[a]_R$, such that

$$[a]_R = \{s \mid (a,s) \in R\}$$

Example Given the set $A = \{a,b,c\}$ and the equivalence relation $R = \{(a,a),(a,b),(b,b),(b,a),(c,c)\}$
Then $[a]_R = \{a,b\}$

Applications of Equivalence

Mathematics Building $\mathbb{R}$, the set of real numbers

Computer Science Traditional C compilers build equivalence classes for variable names

3.7 Partition of a Set

Definition

A partition of a set S is a collection of disjoint non-empty subsets of S that have S as their union. Mathematically, the collection of subsets A_i where $i \in I$ forms a partition of S if, and only if

$$A_i \neq \emptyset \; for \; i \in I$$

$$A_i \cap A_j = \emptyset \; when \; i \neq j$$

$$\bigcup_{i \in I}^{n} A_i = S$$

3.8 Partial Ordering and Posets

Definition A relation R on a set S is called a partial ordering, or partial order, if it is reflexive, antisymmetric and transitive (unlike equivalence relations, which are symmetric).
A set together with a partial ordering R is called a partially ordered set, or **poset**, and is denoted by (S,R).

Example Given $R = \{(a,b) \in \mathbb{Z} \times \mathbb{Z} \mid a \geqslant b\}$
R is a partial ordering on $\mathbb{Z}$ and $(\mathbb{Z}, \geqslant)$ is a poset.

Notation Different poset use different symbols. Therefore, the symbol $\preceq$ is used to symbolise the ordering relation in an arbitrary poset.

Definition The elements a and b of a poset $(S, \preceq)$ are **comparable** if $a \preceq b$ or $b \preceq a$.
When a and b are elements of S so that neither $a \preceq b$ nor $b \preceq a$, then a and b are called **incomparable**.
When a and b are elements of a poset $(S, \preceq)$, it is not necessary that either $a \preceq b$ or $b \preceq a$.

Example $(\mathbb{Z}, |)$ is a poset. However, not all elements are comparable. 3 and 9 or 2 and 4 are comparable but 5 and 7 aren't.

Part III

Algorithms

Chapter 4

What is an algorithm?

An algorithm is a finite sete of well-defined instructions, in order to perform a specific task, for example

- to perform a computation ($x^2 + 3$, $\sum_{i=1}^{n} 4n$)
- to solve a certain problem (Sorting, ordering)
- to reach a certain destination (mostly in maps and graphs[1])

For a little bit of background history, the most ancient proof of an algorithm dates back to 2500BC, in the Babylonian era. It was used to perform a division. The name "Algorithm" comes from Al-Khwārizmī, a Persian polymath who worked on the systematic solving of quadratic equations circa 780AD. However, the most famous mathematician who worked on algorithms remains Alan Turing ($1912 - 1954$), who worked on breaking the Enigma Code, along with the team at Bletchley Park[2].

Specifying Algorithms Algorithms and their set of instructions can be presented in different ways:

- Natural language
- Pseudo-code (non-specific code)
- Programming language (specific code)

Pseudo-code Pseudo-code is an intermediate step between natural language and code. It is precise enough that we know precisely each step but general enough that steps specific to coding (such as variable types or pointers) aren't specified.

This mean of writing an algorithm allows us to analyse the properties of an algorithm independently of any programming language. This is often a useful step in programming, before the implementation of any code.

[1] The most famous problem being the travelling salesman.

[2] If you wanna learn more or enjoy a great film, go watch *The Imitation Game*.

Typical problems

- Searching Problems: finding the position of an element in a list (ordered set).
- Sorting Problems: putting the elements of a list into an increasing order. This can be expanded to other orders and is not limited to numbers (for example, we can sort strings in alphabetical order).
- Optimisation Problems: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.

4.1 Searching Problems

Goal Given a list $S = a_1, a_2, a_3, ..., a_n$ of distinct elements and some x, if $x \in S$, return i such that $a_i = x$. Else, return -1.

Use examples Finding a word in a dictionary, finding a name in a student list, finding an amount in a transaction table.

4.1.1 Linear Search

Definition The linear search algorithm goes through the list, one element at a time, from the first to the last.

$i := 1$
location := 1
while *($i \leq n$ and $x \neq a_i$)* **do**
 | $i = i + 1$
end while
if $i \leq n$ **then**
 | location := i
end if

4.1.2 Binary Search

Definition We assume here that the input is a list of items in **increasing order**.
The algorithms starts by comparing the target value with the middle element of the list. If the middle element is smaller, the algorithm proceeds with the right half of the list. Otherwise, the search proceeds with the left half of the list, including the middle position.
We repeat this process until we have a list of size 1. If our target is equal to the element in the singleton, we return its position. Otherwise, we return 0 (or -1, depends) to indicate that the target was not located.

4.1.3 Linear vs. Binary Search

Linear search:

+ can be applied to *any list*
- is not efficient
- is very slow if the element is not in the list (worst-case scenario)

Binary Search:

+ very efficient on long lists
+ still efficient if the element is not in the list
- all elements must be *comparable*
- list has to be *sorted*

Binary search is more efficient than linear search on long lists because it removes half of the remaining list, instead of only one element per step.

4.2 Sorting Problems

Goal Given a list $S = a_1, a_2, a_3, ..., a_n$, return a list where the elements are sorted in increasing order. (Once again, this can be extended to other orders.)
Sorting is important because a non-negligible part of computing resources are devoted to sorting (for example in large databases).
A great number of fundamentally different algorithms[3] have been invented for sorting and research is still ongoing[4].

4.2.1 Selection Sort

Selection sort makes multiple passes (or iterations) through a list of length n:

- In the first iteration, the minimum of the list is found and put into the first position by swapping the first element with the minimum element.
- Since the first element is now guaranteed to be the smallest after the first pass, we do not take it into account anymore.
- In the second iteration, the minimum of the list from position 2 to n ($\equiv$ the second least element) is found and put into the second position by swapping the second element with the second minimum.
- In the k^{th} iteration, the minimum from position k to n is found and put into the k^{th} position by swapping the k^{th} element with the k^{th} minimum.
- And so on ...

4.2.2 Bubble Sort

Selection sort makes multiple passes through a list:

- In one iteration, every pair of elements that are found to be out of order are swapped (i.e. if $a_i > a_{i+1}$, we swap them).

[3] It can be easier to understand the following algorithms visually. The website `https://mszula.github.io/visual-sorting/` is really helpful for that.

[4] Recently, researches have turned to deep reinforcement learning, as a means to be faster and more efficient.

- Since the last element is now guaranteed to be the largest after the first iteration, in the second iteration, it doesn't need to be inspected.
- In each iteration, one more element at the end becomes sorted and no longer needs inspection, until all elements are sorted.

We can visualise this process as the biggest elements "bubbling" all the way to the end of the list, each one after the other.

4.2.3 Insertion Sort

- We compare the 2^{nd} element with the 1^{st}. If the $2^{nd} < 1^{st}$, we put the 2^{nd} before the 1^{st} : the first two elements are sorted.
- Then, the 3^{rd} element is compared to the 2^{nd}. If $3^{rd} < 2^{nd}$, it is compared to the first one and put in the correct position : the first 3 elements are sorted.
- In each following iteration, the $j{+}1^{st}$ element is put into its correct position amongst the first $j{+}1$ elements.

4.3 Optimisation Problems

Optimization problems minimize or maximize some parameter over all possible inputs.

Examples

- Finding a route between two cities with the smallest total distance (travelling salesman problem)
- Determining how to encode messages using the fewest possible bits (used in .zip compression)

Interestingly enough, the mapping problem can be declined into different problems, according to your priority : fastest or shortest way, fewest connections, least elevation, ... However, these are more complex and precise, and involve more data than we will see during this course.

4.3.1 Greedy Algorithms

Optimization problems can often be solved using a *greedy algorithm*, which makes the "best" choice at each step. This means that it relies on making the locally optimal choice. However, this does not necessarily produce an optimal solution to the overall problem.

Thus, after presenting a greedy algorithm, we either prove that it is the optimal approach or find a counterexample to show that it is not.

One of the principles of greedy programming is that it never reconsiders its choices, contrary to the concept of *dynamic programming*, which is exhaustive but ultimately guaranteed to find the solution.

Funnily enough, greedy algorithms sometimes not only find a bad solution, but produce the unique worst possible solution to certain problems.

4.3.2 Cashier's Algorithm

Problem Find for any amount of n cents, the least total number of coins needed using the following coins : quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent).

(Greedy) Solution At each step, choose the coin with the largest possible value that does not exceed the amount left.

We see here how the choice made is local, because the algorithm does not take into account the amount that will be left after, only what is left at the moment.

Proving Optimality We want to prove that the cashier's algorithm using quarters, dimes, nickels and pennies leads to the optimal solution.

Lemma If $n > 0$, then n cents in change using quarters, dimes, nickels and pennies, using the fewest coins possible,

1. has at most 2 dimes, at most 1 nickel, at most 4 pennies
2. cannot have 2 dimes $(2 * 10¢)$ and 1 nickel $(1 * 5¢)$
3. and the total amount of change in dimes, nickels and pennies cannot exceed 24 cents.

Proof of Lemma 1.1 *If $n > 0$, then n cents in change using quarters, dimes, nickels and pennies, using the fewest coins possible, has at most 2 dimes, at most 1 nickel, at most 4 pennies.*

(a) Dimes, by contradiction : If we have 3 dimes, we have $3 * 10 = 30$ cents, which we replace by 1 quarter $(1 * 25¢)$ and 1 nickel $(1 * 5¢) = 2$ coins $<$ 3 coins.

(b) Nickels, by contradiction : If we have 2 nickels $= 2 * 5 = 10$ cents, we replace them by 1 dime $(1 * 10¢) = 1$ coin < 2 coins.

(c) Pennies, by contradiction : If we have 5 pennies $= 5 * 1 = 5$ cents, we replace them by 1 nickel $(1 * 5¢) = 1$ coin < 5 coins.

Proof of Lemma 1.2 *If $n > 0$, then n cents in change using quarters, dimes, nickels and pennies, using the fewest coins possible, cannot have 2 dimes $(2 * 10¢)$ and 1 nickel $(1 * 5¢)$.*

By contradiction : If we have 2 dimes + 1 nickel = 25 ¢, we replace them with 1 quarter = 25 ¢.

Proof of Lemma 1.3 *Given n cents, with $n > 0$, then the total amount of change in dimes, nickels and pennies cannot exceed 24 cents.*

Per Lemma 1.1, we have at most 2 dimes, 1 nickel and 4 pennies, which is worth 29 cents. Per Lemma 1.2, we cannot have 2 dimes and 1 nickel. If we remove the nickel, which holds the least value, the total is 24 cents.

Proving Optimality Theorem : The greedy change-making algorithm for U.S. coins produces change using the fewest coins possible. Proof by contradiction :

1. Assume that a solution S' exists, is optimal and uses fewer coins than S, the solution produced by the Cashier's Algorithm.
2. let q' be the number of quarters in S'. $q' \leqslant q$ because the cashier's algorithm picks the maximum amount of quarters.
   Can $q' < q$? If $q' < q$, then S' must have $\geqslant 25$ cents with dimes, nickels and pennies. This cannot be optimal, as per Lemma 1.3.
   Thus, $q' = q$.
3. Since $q' = q$, S and S' have to change the same remaining amount of money by using dimes, pennies and nickels.
4. $d' \leqslant d$ because the cashier's algorithm takes the maximum possible amount of dimes.
   Can $d' < d$? If $d' < d$, we need at leasts 2 extra nickels to make up for the value of the missing dime. This cannot be optimal, due to Lemma 1.1.
   Thus, $d' = d$.
5. Since $q' = q$ and $d' = d$, S and S' have to change the same remaining amount of money using nickels and pennies.
6. $n' \leqslant n$, because the cashier's algorithm takes the maximum possible amount of nickels.
   Can $n' < n$? If $n' < n$, then S' would have at least 5 extra pennies to make up for the missing nickel(s). This cannot be, as per Lemma 1.1.
   Thus, $n' = n$.
7. Since $q' = q$, $d' = d$ and $n' = n$, the remaining amount of money is the same for S and S'. They need the same amount of pennies, leading to $p' = p$.
8. We assumed S' to be optimal, so S is optimal too.

Remark It is important to note that we have only proven that the Cashier's Algorithm is optimal when using the American coin system. Using another set of coins does not guarantee that the greedy solution is optimal, we would have to prove that.

4.4 Matching and Stable Matching

Goal Pair elements from two equally sized groups considering their preferences for members of the other group so that there are no ways to improve the matching (according to the preferences).

Matching Given a finite set A, a *matching* of A is any set of unordered pairs of *distinct* elements of A where any element occurs in at most one pair. (Such pairs are called *independent*.) This mean that any element can appear at most once per matching set.

Maximum matching A *maximum matching* is a matching that contains the largest possible number of independent pairs.

Preference list A *preference list* L_x defines for every element $x \in A$ the order in which the element prefers to be paired with another element.

Example Let L_x be the preference list of x. If $L_x = [a, b, c]$, x would rather be paired with a, then, if not possible, with b, then c.

Stability and Instability A matching is *unstable* id there are two pairs $(a, b), (c, d)$ in the matching such that a prefers c over b and c prefers a to d. This means that we can exchange to elements between pairs and get a better match, according to their preferences.
A *stable* matching is a matching that is not unstable. This means that there is no pair of participants that prefer each other to their assigned match.

Example Given the set $A = \{Peter, Dana, Egon, Ray\}$ and the preference lists

- $L_{Peter} = [Dana, Egon, Ray]$
- $L_{Dana} = [Peter, Egon, Ray]$
- $L_{Egon} = [Peter, Dana, Ray]$
- $L_{Ray} = [Peter, Dana, Egon]$

The matching $\{(Ray, Dana), (Egon, Peter)\}$ is unstable, because we can exchange Peter and Ray (Peter and Dana prefer each other to their assigned matches). Therefore, the matching $\{(Peter, Dana), (Egon, Ray)\}$ is stable.

4.4.1 Marriage Problem

Definition Given a set A with even cardinality (= even number of elements), partition A into two disjoint subsets A_1 and A_2 with $A_1 \cup A_2 = A$ and $|A_1| = |A_2|$. A matching is a bijection from the element of one set to the element of the other set.
That means that pairs can only consist of one element of A_1 and A_2 each.

Goal Find a *maximum stable matching* for $A_1 \cup A_2 = A$. This is called the marriage problem.

Gale-Shapley Algorithm This is a greedy algorithm, also known as the deferred acceptance algorithm and propose-and-reject algorithm, used to construct a stable maximum matching to answer the marriage problem.

This algorithm is named after David Gale and Lloyd Shapley, who proved in 1982 that it is always possible to find a stable matching that answers the marriage problem. An very similar algorithm has been used since the 1950s to match medical school students to residency programs across the U.S.

Algorithm 1: Gale-Shapley Algorithm

```
Let M be the set of pairs under construction
Initially M = ∅
while | M |<| A_1 | do
    Select an unpaired x ∈ A_1
    Let x propose to the first element y ∈ A_2 on L_x
    if y is unpaired then
        Add the pair (x, y) to M
    else
        Let x' ∈ A_1 be the element that y is paired to (i.e. (x', y) ∈ M
        if x' precedes x on L_y then
            Remove y from L_x
        else
            Replace (x', y) ∈ M by (x, y) and remove y from L_x'
        end if
    end if
end while
```

4.5 Unsolvable Problems

Can every problem be solved by an algorithm? This question was solved by Turing and the answer is "No". To prove that, Turing defined an unsolvable problem : the *halting problem.*

The Halting Problem

Can we develop a procedure that takes as input a computer program and an input and determines whether the program will eventually finish running or continue to run forever?

This problem is unsolvable because, as long as the program is running, we cannot determine if it will stop or not. More particularly, if a program halts, we can determine that it does in fact halt. But if it never halts, we cannot determine whether it will halt or not, until it halts, which it will not.

Decidability An *undecidable* problem is a decision problem for which it has been proven that no algorithm can always output a correct yes-or-no answer.

4.6 Efficiency of algorithms

In order to compare different algorithms, we have to different extreme precision degrees:

1. Precise count of everything involved (such as computer instructions, disk accesses, ...)
   This means that the efficiency would be a function the size of the problem, which is inconvenient and not always well-defined.

2. "It took a few seconds on my laptop"
   That statement is not informative. What if you run it at NASA? And

what about on a Smaky 4? Then, the time and resources needed are not predictable.

Therefore, we use another way of estimating the efficiency of an algorithm (or any procedure or computation), that is the next chapter of this class.

Chapter 5

Growth of Functions and Algorithm Complexity

Introduction Imagine that you run the following algorithm:

Algorithm 2: *Sort_tasks*(n : *integer*)
Create a list of 3000 random numbers and sort it using bubble sort (task 1)
Create n lists of length 1500 and sort them using bubble sort (task 2)
Create a list of length $400 * n$ and sort it using bubble sort (task 3)

We measure how much time is spent on each task depending on n ($n = 1, 2, 3, ..., 20$) and we find that it is approximately

- 1000 milli-seconds for task 1, independent of n
- $200 \cdot n$ milli-seconds for task 2
- $1.5 \cdot n^2$ milli-seconds for task 3

Now, we can estimate the time spent for all tasks as a function of n: $f(n) = 1.5n^2 + 200n + 1000$

Let $g(n) = 1.5 \cdot n^2$, $h(n) = 200 \cdot n$ and $j(n) = 1000$, such that $f(n) = g(n) + h(n) + j(n)$.

We notice that for a small value of n, $j(n)$ holds the most significant value. Then, $h(n)$ takes over and, ultimately, only $g(n)$ is relevant.

Observation If we generalize that observation, let $f(n)$ estimate the time to solve a problem of size n. If $f(n) = g(n)+h(n)+...+j(n)$ for functions g,h,...,j: $\mathbb{N} \to \mathbb{R}$. Then, the "ultimately largest" of g,h,...,j determines the behaviour of f as n grows.

Let $g(n)$ be the "ultimately most relevant part" of $f(n)$. Then, the growth rate of $f(n)$ is independent of multiplicative constants in $g(n)$: $\frac{g(m)}{g(n)} = \frac{c \cdot g(m)}{c \cdot g(n)}$

With all of those observations, we can deduce than when considering a runtime function (here $f(n)$), we

- only focus on the part that grows the "fastest" (as we care about the cases where $n \to \infty$)
- forget about any multiplicative constant, because the absolute value is not important here, rather than the growth

only focus on the part that grows the "fastest" (for $n \to \infty$)

5.1 Big-O Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ **is** $\mathcal{O}(g(x))$ if there exist constants C and k such that

$$| f(x) | \leq C | g(x) |, \ whenever \ x > k$$

This is read as "$f(x)$ is big-O of $g(x)$" or "*g is an asymptotical bound (above) for f*" which is a complicated way to say that it is an upper bound (or limit) on the growth rate of f.

The constants C and k are called witnesses to the big-O relationship. Only one pair of witnesses is needed. However, to show that $f(x)$ is *not* big-O of $g(x)$, we have to show that we cannot find such a pair. This can be done using a proof by contradiction.

Example and common uses

75 is $\mathcal{O}(1)$	and 1 is $\mathcal{O}(75)$
$cos(x)$ is also $\mathcal{O}(1)$	and $sin(x)$ is $\mathcal{O}(1)$
1 is $\mathcal{O}(x)$	but x is not $\mathcal{O}(1)$
x is $\mathcal{O}(x^2)$	but x^2 is not $\mathcal{O}(x)$
x^2 is $\mathcal{O}(x^2)$	and x^2 is $\mathcal{O}(x^3)$
x^2 is $\mathcal{O}(6x^2 + x + 3)$	and $6x^2 + x + 3$ is $\mathcal{O}(x^2)$

However, We rarely use $\mathcal{O}(6x^2+x+3)$ and $\mathcal{O}(75)$ because our goal is to simplify, so that it is easy to grasp the growth rate of a function.

5.1.1 Big-O Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x^1 + A_0$ where $a_0, a_1, ..., a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is $\mathcal{O}(x^n)$. The leading term $a_n x^n$ of a polynomial dominates its growth.

It follows that if a polynomial is $\mathcal{O}(x^n)$, then its degree is $\leqslant n$.

5.1.2 Combinations of Functions

Many algorithms are made up of two or more separate sub-procedures. The number of steps used by a computer to solve a problem with an input of a specified size n using such an algorithm is the sum of the number of steps used by these sub-procedures.

Sum of functions

Theorem: Suppose that $f_1(x)$ is $\mathcal{O}(g_1(x))$ and that $f_2(x)$ is $\mathcal{O}(g_2(x))$. Then $(f_1 + f_2)(x)$ is $\mathcal{O}(g(x))$ where $g(x) = max(|g_1(x)|, |g_2(x)|)$ for all x.

Corollary: Suppose that $f_1(x)$ and $f_2(x)$ are both $\mathcal{O}(g(x))$. Then $(f_1 + f_2)(x)$ is $\mathcal{O}(g(x))$.

Product of functions

Theorem: Suppose that $f_1(x)$ is $\mathcal{O}(g_1(x))$ and that $f_2(x)$ is $\mathcal{O}(g_2(x))$. Then $(f_1 \cdot f_2)(x)$ is $\mathcal{O}(g_1(x) \cdot g_2(x))$.

Transitivity

Suppose that $f(x)$ is $\mathcal{O}(g(x))$ and $g(x)$ is $\mathcal{O}(h(x))$. Then $f(x)$ is $\mathcal{O}(h(x))$.

5.1.3 Useful Big-O Estimates

- n^c is $\mathcal{O}(n^d)$ but n^d is not $\prime(n^c)$ for $d > c > 1$
- $(log_b n)^c$ is $\mathcal{O}(n^d)$, but n^d is not $\mathcal{O}((log_b n)^c)$ for $b > 1, c, d > 0$
- n^d is $\mathcal{O}(b^n)$ but b^n is not $\mathcal{O}(n^d)$ for $d > 0, b > 1$
- b^n is $\mathcal{O}(c^n)$ but c^n is not $\mathcal{O}(b^n)$ for $c > b > 1$
- c^n is $\mathcal{O}(n!)$ but $n!$ is not $\mathcal{O}(c^n)$ for $c > 1$

Notation If we want to write a big-O relation with mathematical signs, we can write $f(x) \in \mathcal{O}(g(x))$, because $\mathcal{O}(g(x))$ represents the set of functions that are $\mathcal{O}(g(x))$.

Summary of Big-O Notation and nomenclature

- Constant: $\mathcal{O}(1)$
- Logarithmic: $\mathcal{O}(logx)$
- Poly-logarithmic: $\mathcal{O}((logx)^d), d > 1$
- Linear: $\mathcal{O}(x)$
- Linearithmic: $\mathcal{O}(xlogx)$
- Polynomial: $\mathcal{O}(x^d), d > 1$
- Exponential: $\mathcal{O}(b^x), b > 1$
- Factorial: $\mathcal{O}(x!)$

5.2 Big-Omega Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ **is** $\Omega(g(x))$ if there are constants $C > 0$ and k such that

$$| f(x) | \geq C | g(x) | \text{ whenever } x > k$$

This is read as "$f(x)$ is big-Omega of $g(x)$". Big-Omega tells us that a function grows *at least as fast* as another (big-Omega is the lower bound to the growth of a function).

$f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $\mathcal{O}(f(x))$

5.3 Big-Theta Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ **is** $\Theta(g(x))$ if $f(x)$ is $\mathcal{O}(g(x))$ and $f(x)$ is $\Omega(g(x))$

This is read as "$f(x)$ is big-Theta of $g(x)$" or $f(x)$ is of *order* $g(x)$".

$f(x)$ is $\Theta(g(x))$ if and only if there exist positive constants C_1, C_2 and k such that

$$C_1 | g(x) | \leq | f(x) | \leq C_2 | g(x) | \text{ if } x > k$$

When $f(x)$ is $\Theta(g(x))$ then $g(x)$ is $\Theta(f(x))$

5.3.1 Big-Theta Estimates for Polynomials

Theorem Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x^1 + A_0$ where $a_0, a_1, ..., a_n$ are real numbers with $a_n \neq 0$.

We already saw that the leading term $a_n x^n$ of a polynomial dominates its growth. Then $f(x)$ is of order x^n or $f(x)$ is $\Theta(x^n)$.

5.4 Little-o Notation

We say that $f(x)$ **is** $o(g(x))$ if

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$

We also say the "f is little-o of g" or "f is little-omicron of g"

Examples

x^2 is $\mathcal{O}(x^3)$ and it is also $o(x^3)$ as $\lim_{x\to\infty} \frac{x^2}{x^3} = 0$

$x^2 + x + 1 is \mathcal{O}(x^2)$ but it is not $o(x^2)$ as $\lim_{x\to\infty} \frac{x^2+x+1}{x^2} = 1$

In general, we can say that x^n is $o(x^{n+1})$ $\forall x \geq 0$ because $\lim_{n\to\infty} \frac{x^n}{x^{n+1}} = 0$

5.4.1 Little-o and Big-O

If $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$, then $f(x)$ is $\mathcal{O}(g(x))$

However, if $f(x)$ is $\mathcal{O}(g(x))$, it does not necessarily follow that $f(x)$ is $o(g(x))$.

$f(x) \in o(g(x)) \implies f(x) \in \mathcal{O}(g(x))$

Theorem: finite limit and Big-O

If $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c$ with c finite, then $f(x) \in \mathcal{O}(g(x))$

This is proved as follows:

1. $\forall \varepsilon > 0, \forall x > k, |\frac{f(x)}{g(x)} - c| \leqslant \varepsilon$ (definition of the limit)
2. $|\frac{f(x)}{g(x)}| = |\frac{f(x)}{g(x)} - c + c| \leqslant |\frac{f(x)}{g(x)} - c| + |c| = \varepsilon + |c|$
3. We take $\varepsilon = 1$ (but we can pick any value we want). $|\frac{f(x)}{g(x)}| \leqslant 1 + |c|$.
4. It follows that $|f(x)| \leqslant (1 + |c|)\ |g(x)|$

5.5 Back on Algorithm Complexity

Given an algorithm, we want to measure how efficient it is for solving a problem given an input of a particular size (*computational complexity*). For that, we differentiate two types:

- **Time complexity**: How much time is needed to solve the problem of an input of a given size
- **Space complexity**: How much computer memory is used

Understanding complexity is important to understand whether it is practical to use an algorithm for inputs of a particular size and to compare the efficiency of different algorithms designed to solve the same problem.

This course focuses only on time complexity, which corresponds to the number of operations performed is the algorithm is sequential. To describe that, we will use big-O and big-Theta notation. We will also ignore implementation details, because that is too complicated for the scope of this course.

To determine time complexity, we need to determine the number of basic operations (such as addition, multiplication, comparisons, swaps, etc.) while ignoring details like the "house keeping" aspects of the algorithm (storing a value in a variable, incrementing a variable, etc.).

5.5.1 Worst-case Time Complexity

Sometimes, the number of operations depends on the exact configuration of the input (a position in a list, ...), which is why we focus on the *worst-case* time complexity of an algorithm. This provides an upper bound on the number of operations depending on the input size. It is generally much more difficult to determine the *average case* time complexity, which is the average number of operations for all inputs of a particular size.

5.5.2 Complexity of important algorithms

Worst case complexities

- Linear Search: $\Theta(n)$
- Binary Search: $\Theta(log_2(n))$

- Bubble Sort: $\Theta(n^2)$
- Insertion Sort: $\Theta(n^2)$
- Selection Sort: $\Theta(n^2)$

5.5.3 Effect of Complexity

A single operation takes around 10^{-9} seconds, meaning we perform 1 billion bit operations per second (on a processor running at 1 GHZ).

The bigger the time complexity of an algorithm, the longer it takes to run on any computer. For example, an algorithm that is $\Theta(n \cdot log(n))$ takes $2 \cdot 10^{-2}$ seconds for an input of size 10^6 but an algorithm of complexity n^2 would take 17 minutes. That difference does not seem like much when we only use algorithms with small input sizes in our daily lives but is is crucial in companies using large databases.

5.5.4 Complement: Decreasing Complexity

This was not seen during class, but I thought it interesting to add as a complement.
Consider the following problem:

Given a sequence (a_n), you want to find a value j such that there exists an element a_i of the sequence that in a given range $[a, b]$ for $i < j$. This means that, the greater the value of $b - a$, the less you will have to compute the sequence to find a value in that range.

For example, if we take the sequence $a_n = n \cdot sin(n)$, we will have (a_n) between n and $-n$. If we want to get a value between a small range $[a, b]$, we will have to compute a lot of values of (a_n). However, for a bigger range, it is much more likely that there will be a value within that range early on. In this example, the complexity is $\frac{1}{b-a}$.

Chapter 6

Induction, Recursion

Induction and recursion are different approaches to proving results and solving problems.

They both in the first place rely on the ability to achieve the desired result for the smallest possible version of the problem at hand.

Induction extends this ability to problems of any size, while recursion reduces a problem of any size to the smallest possible ones.

Definition A *recursive definition* of a function f with the set of non-negative integers as its domain consists of two steps:

Basis step: specify the value of f at 0.

Recursive step: Give a rule for finding its value at $f(n)$, with n an integer, from its values at smaller values of n.

A function $f(n) : \mathbb{Z}_{\geqslant 0} \to \mathbb{R}$ is the same as a sequence $a_0, a_1, a_2, ...$ where a_i is a real number for every $i \in \mathbb{Z}_{\geqslant 0}$[1].

Recursive functions can be seen as a more generalized general version of recursive sequences, because their domain of definition is not limited to integers.

Example The factorial function $n!$ can be defined recursively as $f(0) = 1$ (basis step) and $f(n+1) = (n+1) \cdot f(n)$.

The Fibonacci numbers are defined as

$f_0 = 0$

$f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$

6.1 Recursion on Strings

Definition The set Σ^* of strings over the alphabet Σ is defined inductively by

Basis step: $\lambda \in \Sigma^*$ (where λ is the empty string)

Inductive step: If $w \in \Sigma^*$ and $x \in \Sigma$ (x is a character), then $wx \in \Sigma^*$

The recursive definition of $l(w)$, the function returning the length of the string w is:

[1] aka a "*suite*" (a_n)

Basis step: $l(\lambda) = 0$

Recursive step $l(wx) = l(w) + 1$ if $w \in \Sigma^*$ (a sequence of characters) and $x \in \Sigma$ (a single character).

Theorem $l(xy) = l(x) + l(y)$, where x and y belong to Σ^*, the set of strings over the alphabet Σ.
Here the induction proof is not over $P(n)$ but over $P(x, y)$. In order to do the proof, we have to focus on one parameter only: $P(x, y)$ becomes $P(y) = \forall x \in \Sigma^*, l(xy) = l(x) + l(y)$

- Basis step: we always use $\lambda =$ "" as the base case for strings
  $\forall x \in \Sigma^*, l(x\lambda) = l(x) + l(\lambda) = l(x)$
- Inductive step: $P(y) \rightarrow P(yc)$ is true $\forall c \in \Sigma$ (c is a character)
  $P(yc) = \forall x \in \Sigma^*,\ l(xyc) = l(xy) + 1$ (by definition of recursive l)
  $= l(x) + l(y) + 1$ (by induction hypothesis)
  $= l(x) + l(yc)$ (by definition of recursive l)

6.2 Recursive Algorithms

An algorithm is called *recursive* if it solves a problem by reducing it to an instance of the same problem with a smaller input.

For the algorithm to terminate, the instance of the problem must eventually be reduced to some initial case for which the solution is known.

The first step is to find a recursive definition of the problem (or function).

The second step is to write a method (using pseudocode first) that solves the problem using the recursive definition

Example To compute the factorial $n!$ of an integer $n \geqslant 0$, we do the following:

- Step 1: recursive definition of factorial function $n!$
- Step 2: recursive procedure for computation of factorial

Procedure factorial(n: non-negative integer)

```
if n = 0 then
 | return 1
else
 | return n · factorial(n − 1)
```

6.2.1 Intermezzo: Call Stack

A call stack stores information about the active subroutines of a program and works kind of like a pile of books:

- the caller pushes the return address onto the stack
- the called subroutine, when it finished, pulls or pops the return address off the call stack and transfers control to that address

An active subroutine is one that has been called but is yet to complete execution, after which control is handed back to the caller (or point of call). One common error in programming is that of `StackOverflow Error`. This means that all the space allocated for the call stack has been consumed, generally by an infinitely recursive function.

Example If we take once again our example of the recursive factorial with a value of $n = 6$, we first add to the call stack `factorial(6)`. The function becomes `return 6·factorial(5)` and then *calls* `factorial(5)`. This goes on until `factorial(2)` becomes `return 2·factorial(1)`. This finally calls `factorial(1)`, which returns 1. The control is then return to `return 2·factorial(1)` and so on, until the stack is empty.

6.3 Recursion and Iteration

We can evaluate a recursively defined function using a:

- Recursive algorithm: applies directly the recursive definition until the base is reached
- Iterative algorithm: starts with the base cases and applies the recursive definition to compute the function for larger values

Example Our goal is to compute a^n where $a \in \mathbb{R}^*$ and $n > 0$

Procedure power recursive(a, n)

if *n=0* **then**
  return 1
else
  return $a \cdot power(a, n-1)$

Procedure power iterative(a, n)

res = 1
for $i = 1$ **to** n **do**
  res = a*res
return res

6.3.1 Divide and Conquer

Strategy for solving a problem of size n:

1. **Divide**:
   - if $n > 1$: divide the problem of size n into 2 (almost) equally sized subproblems
   - else: solve the problem of size 1 directly

2. **Conquer**: solve the subproblems in the same way (recursively)
3. **Merge**: merge the subsolutions into an overall solution

Example: Binary Search Assume we have $a_1, a_2, ..., a_n$ an increasing sequence of integers. We are looking for an element x in this sequence.

1. Divide:
   (a) if $n > 0$: find the middle element a_m of the list
   (b) else: the sequence is empty, x is not in the list
2. Conquer:
   (a) if $x = a_m$: we have found the element x and can stop the search here
   (b) if $x < a_m$: search the left half (up to a_{m-1})
   (c) if $x > a_m$: search the right half (from a_{m+1} up)

The pseudocode for that algorithm would look something like this: We can see

Procedure binary search(x,l,r: integers)

```
if l > r then
    return -1
else
    m:=⌊(l + r)/2⌋
    if x < a_m then
        binary_search(x, l, m − 1)
    else if x > a_m then
        binary_search(x, m + 1, r)
    else
        return m
```

that each time, the length n of the sequence is divided by 2. This corresponds to a complexity of $\mathcal{O}(log(n))$.

Sorting algorithms like Bubble Sort previously had a complexity$\in \Theta(n^2)$. Now, Merge Sort is significantly faster in sorting a sequence S of size n.

6.3.2 Merging two sorted lists

Traverse the two lists simultaneously from left to right. Always take the smaller element of the two lists.

Procedure merge(L_1, L_2: sorted lists)

```
L := empty lists while L_1 and L_2 are both non-empty do
    remove smallest of first elements of L_1 and L_2 from its list
    put it at the right end of L
    if this removal make one list empty then
        remove all elements from the other list and append them to L
return L
```

The worst case for this procedure is when the two list alternate, because we have to keep comparing elements. This means $|L_1| + |L_2| - 1$ comparisons.

Merge Sort Complexity Merge sort works as follows:

Procedure mergesort($L = a_1, a_2, ..., a_n$)

if $n > 1$ **then**
- $m := \lfloor n/2 \rfloor$
- $L_1 := a_1, a_2, ..., a_m$
- $L_2 := a_{m+1}, a_{m+2}, ..., a_n$
- $L := merge(mergesort(L_1), mergesort(L_2))$

Merge sort works in layers (k). We set $n = 2^m \Leftrightarrow m = log_2(n)$.

At $k = 0$, we have 1 list of length $2^m = n$.

At $k = 1$, we have 2 lists of length $2^{m-1} = \frac{n}{2}$.

...

At $k = m$, we have 2^m lists of length 1.

Now, we go backwards and merge lists

At $k = m$ we need to do 2^{m-1} merges between lists of size 1, with a complexity of range $1 + 1 - 1 = 2 - 1$

At $k = m - 1$, we have 2^{m-1} lists and do 2^{m-2} merges between lists of size 2, with a complexity range of $2 + 2 - 1 = 4 - 1$

If we generalize, for any k, we have 2^k lists and 2^{k-1} merges to do, which is a complexity of $2^{m-k} + 2^{m-k} - 1 = 2^{m-k+1} - 1 = 2^{m-(k-1)} - 1$

The total complexity is

$$\sum_{k=1}^{m} \overbrace{2^{k-1}}^{nb.\ merges} \cdot (2^{m-(k-1)} - 1) = \sum_{k=1}^{m} (2^{m-(k-1)+(k-1)} - 2^{k-1})$$

$$= m \cdot 2^m - \sum_{k=1}^{m} 2^{k-1}$$

$$= m \cdot 2^m - \underbrace{\frac{1 - 2^m}{1 - 2}}_{negligible} \in \mathcal{O}(m \cdot 2^m) = \mathcal{O}(n log_2(n))$$

6.4 Number Representation

6.4.1 Representation of Integers

In general, we use decimal notation to represent integers. For example, 965 represents 9 hundreds + 6 tens + 5 ones or $9 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0$. We also call this *base 10* notation.

However, different contexts call for different representations. For example, computers use binary numbers or base 2 notation.

We can represent numbers using any base b where b is a positive integer greater than 1. As an example, the ancient Mayans used a base 20 and the ancient Babylonians used base 60.

The ones we will use most in computing and communications are bases $b = 2$(binary), $b = 8$(octal) and $b = 16$(hexadecimal).

6.4.2 Base b representation

For a base b (b ¿ 1) and any positive integer n, there is a unique choice of

- k, a non-negative integer
- $a_0, a_1, ..., a_k$ integers with $0 \leq a_j \leq b-1$ and $a_k \neq 0$

$$n = a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b + a_0$$

The a_j coefficients are called the base b digits of the representation

6.4.3 Base b expansion of n

For a positive integer n, when $n = a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b + a_0$, we write

$$(a_k a_{k-1} ... a_1 a_0)_b$$

the representation in base b of n.

6.4.4 Common Bases

- Decimal Expansion base 10
- Binary Expansion base 2
- Octal Expansion base 8
- Hexadecimal Expansion base 16

Example $(17)_{10} = 1 \cdot 10^1 + 7 \cdot 10^0 = (10001)_2 = 1 \cdot 2^4 + 1 \cdot 2^0 = (21)_8 = 2 \cdot 8^1 + 1 \cdot 8^0 = (11)_{16} = 1 \cdot 16^1 + 1 \cdot 16^0$

Binary Expansions Most computers represent integers and do arithmetic with binary expansion of integers, with 0 and 1 being the only digits used (representing true and false).

Octal Expansions The octal expansion uses the digits $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Hexadecimal Expansions

Because we only have the digits 0 to 9 but need 16 digits for the representation, we use letter to expand our usual digits set. Thus $10 = A, 11 = B, 12 = C, 13 = D, 14 = E$ and $15 = F$ (and 16 is simply $1 \cdot 16^1$ because the digits must be smaller than the base).

The digits are then $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

6.4.5 Constructing a base b expansion

What is $(n)_10$ in binary notation?

One algorithm would be to first find k and then work down to find a_k then a_{k-1} etc. This would be a version of the cashier's algorithm but with the base instead of the coin values.

Algorithm:

- Find k by computing successive powers of b until you find the smallest k such that $b^k \leq n < b^{k+1}$
- For each value of i from 0 to k:
  - Set a_{k-1} to be the largest number between 0 and $b-1$ for which $a_{k-1} \cdot b^{k-1} \leq n$
  - Update the current remaining value $n := n - a_{k-1} \cdot b^{k-1}$

6.4.6 Bases and Divisibility

When $k > 0$:

$$\begin{aligned} n &= a_k b^k + a_{k-1} b^{k-1} + ... + a_1 b + a_0 \\ &= b(a_k b^{k-1} + a_{k-1} b^{k-2} + ... + a_1) + a_0 \end{aligned}$$

If a and b are integers with $a \neq 0$, then a divides b if there exists an integer c such that $b = a \cdot c$ or $\frac{b}{a}$ is an integer

Notation The notation $a \mid b$ denotes that a divides b. If a does not divide b, we write $a \nmid b$.

6.4.7 Algorithm

Theorem If a is an integer and d is a positive integer, then there are unique integers q and r (with $0 \leq r < d$) such that $a = d \cdot q + r \Leftrightarrow \frac{a}{q} = d + \frac{r}{q}$

d is called the *divisor*.
a is called the *dividend*.
q is called the *quotient*.
r is called the *remainder*.

For $a = d \cdot q + r$ we write:

$q = a$ **div** d div is a function: **div**:$\mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{Z}$
$r = a$ **mod** d mod is a function: **mod**:$\mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{N}$

Base b Expansion Algorithm
$(a_{k-1}, ..., a_1, a_0)$ is the base b expansion of n. The digits are the remainders of the division given by q $\pmod b$.

Procedure base b expansion(n, b: positive integers)

$q := n$
$k := 0$
while $q \neq 0$ **do**
  $a_k := q \pmod b$
  $q := qdivb$
  $k := k + 1$
return $(a_{k-1}, ..., a_1, a_0)$

Example We are trying to compute $(12'345)_8$

$$12'345 = 8 \cdot \overbrace{1'543}^{remainder} + 1$$
$$\overbrace{1'543} = 8 \cdot 192 + 7$$
$$192 = 8 \cdot 24 + 0$$
$$24 = 8 \cdot 3 + 0$$
$$3 = 8 \cdot 0 + 3$$
$$(30071)_8 = 12'345$$

Part IV

Counting

Chapter 7

Basic Counting Principles

7.1 Counting Subsets of a finite set

We use the product rule to show that the number of different subsets of a finite set S is $2^{|S|}$.
Proof:

- When the elements of S are listed in an arbitrary order, there is a one-to-one correspondence between subsets of S and bit strings of length $|S|$.
- When the i^{th} element is in the subset, the bit string has a 1 in the i^{th} position and a 0 otherwise.
- By the product rule, there are $2^{|S|}$ such bit strings, and therefore $2^{|S|}$ subsets.

Example $A = \{a, b, c, d\}$
We can represent any subset of A by a bitstring of n times 0 and 1. Here, 1101 corresponds to the subset $\{a, b, d\}$ of A.
We can see that there are $2 \times 2 \times ... \times 2 = 2^n$ possible bitstrings. Hence, the cardinality of the power set of S is 2^n.

7.2 Counting functions

How many functions are there from a set with m elements to a set with n elements?

Let $m = \{0, 1, 2, ..., m-1\}$. Each element of m can be mapped to any element of n, meaning there are n possibilities for every element of m, which leads to $n \times n \times ... \times n = n^m$ possibilities $= n^m$ functions.

7.2.1 Counting one-to-one functions

How many *injective functions* are there from a set with m elements to a set with n elements?

Let $m = \{0, 1, 2, ..., m-1\}$. The first element can be mapped to n elements of n. The second can be mapped to $n-1$ elemnts of n, because the functions

are injective and one element of n is already mapped to by one element of m. This goes on until the m^{th} element of m can be mapped to the only remaining element of n. Thus, our possibilities are $n \cdot (n-1) \cdot (n-2) \cdot ... \cdot (n-n+1)$.

7.3 The Sum Rule

If $S_1, S_2, ..., S_n$ are finite **disjoint** sets, then $|\bigcup_{i=1}^{n} S_i| = |S_1| + |S_2| + ... + |S_n|$.

Example Counting Passwords: Each user on a computer system has a password, which is 6 to 8 characters long, where each character is an uppercase letter or a digit. Each password must contain at least one digit.
How many possible passwords are there?

To do this, we separate the passwords into three distinct sets P_6, P_7, P_8 depending on the number of characters. Then, for each set, we count all the possibilities minus the possibilities without any digit.
This is counted as $P_6 = (26+10)^6 - 26^6$, $P_7 = 36^7 - 26^7$, $P_8 = 36^8 - 26^8$. Then, because the three sets are disjoint, we add them: $P = P_6 + P_7 + P_8$