AICC 1 - Notes and Summary

Faustine Flicoteaux

Fall Semester 2024

Introduction

These are my notes for the Advanced Information, Communication and Computation I course given during the fall semester of 2024.
They are not be exempt of errors. If you find one, please contact me at $faustine.flicoteaux@epfl.ch$.

Chapter 1

Propositional Logic

This is some text

Chapter 2

Predicate Logic

This is some other text

Chapter 3

Relations

3.1 Binary relations

Definition A binary relation R from a set A to a set B is a subset 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 $a \in A$.

3.2 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 $R = \{(a, a) | a \in A \ \forall a\}$

3.3 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, $\forall a \forall b \forall c((a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$.

3.4 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.

3.5 Combining Relations

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

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

3.6 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 $S \circ R$.

3.7 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

3.8 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

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

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

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

3.9 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.

Chapter 4

Algorithms

4.1 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.

4.2 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.

4.2.1 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

4.2.2 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.

4.2.3 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.

4.3 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 have been invented for sorting and research is still ongoing[3].

4.3.1 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 ...

4.3.2 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).
- 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.

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

- 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.

4.3.3 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.

4.4 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.

4.4.1 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.

4.4.2 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.

4.5 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.

4.5.1 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
```

4.6 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.

4.7 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

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 $n \to \infty$)

Chapter 6

Incursion, 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.