Data Structures and Algorithms–Lecture Notes

Winter 2024

Jelle Hellings

Department of Computing and Software
Faculty of Engineering, McMaster University
1280 Main Street West, Hamilton, ON L8S 4L7, Canada

Chapter 1

Introduction

In these notes, we focus on the study of *data structures* and *algorithms*, two closely related topics. These topics are central to computer science, as data structures and algorithms are the basic building blocks of any *correct* and *efficient* solution for problems that can be solved by a computer program.

First, we shall briefly describe what an algorithm is.

Definition 1.1. An *algorithm* is a description on how to solve a specific problem that is suitable for implementation via a program. Typical algorithms takes one-or-more input values and, based on these inputs, produce some output via a well-defined computational procedure.

Take, for example, the ArraySum algorithm of Figure 1.1. This *algorithm* solves the problem of computing the sum of an array L of values. It does so via a standard while loop that visits each value in the array L and adds them to *sum*.

Algorithm ArraySum(L) :
Input: L is an *array* with $N = |L|$ values $L[0], \ldots, L[N-1]$.
Result: return the value $\sum_{v \in L} v = L[0] + \cdots + L[N-1]$.

```
1: sum, i := 0, 0.
2: while i < N do
3:     sum, i := sum + L[i], i + 1.
4: end while
5: return sum.
```

Figure 1.1: An *algorithm* for computing the sum of array L.

Given the ArraySum algorithm, we can ask a few basic questions about its workings:

1. Is this algorithm *correct*?
2. Is this algorithm *efficient*? Can we do *better* or is it *optimal*?

Intuitively, the ArraySum looks correct and efficient. It is even likely you wrote programs with similar functions before and, after debugging and testing, you determined those programs work. Unfortunately, debugging and testing does not provide *guarantees*: you cannot test ArraySum with *every* array in existence. In these notes, we look at how we can *formally* argue whether an algorithm is correct and efficient.

Remark 1.2. In these notes, we do not write algorithms in a specific programming language: the choice of programming language *does not really* matter. Instead, we typically use a pseudo-code notation such as the one used in Figure 1.1. The usage of pseudo-code allows us to focus on the important bits, without having to delve into the specifics on how to implement details in a specific programming language. In the few examples in which we provide real source code, we shall do so in C++. Knowledge of C++ is not necessary to follow these examples, however.

Algorithms can be seen as an *abstraction* of a function, method, or procedure in your program: in this sense, the ArraySum algorithm of Figure 1.1 is very *detailed*: one can copy-and-paste the algorithm into a program with only minor changes in notation. We only provide this level of detail in some of the algorithms we describe.

Most interesting algorithms operate on large collections of data, e.g., ArraySum operates on an array of values. Hence, the study of algorithms goes hand in hand with the study on how to store and manipulate large collections of data:

Definition 1.3. A *data structure* is a scheme describing how to store and organize data in order to facilitate efficient access to and modification of the data. Typically, data structures consist of an *internal representation* of the data and *operations* to change the data (e.g., adding, removing, or updating data)

A very simple example of a *data structure* is an array: an array A holds a list of $N = |A|$ values and allows one to easily read or update the i-th value $A[i]$, $0 \leq i < N$. In most programming languages, the *internal representation* of array A is a *block of memory* that can hold N consecutive values. For example, a C++ array `std::array<int, 12> L` is an array that can hold 12 values of type `int`. Here, we assume that a single `int` value is a 32-bit value that requires 4 byte to store. In Figure 1.2, we illustrate how this array will be stored in memory. Using this internal representation, we can access `L[5]` (the *sixth* value in this array) by simply accessing the `int` value starting after the 20-th byte in the memory *representing* the array.

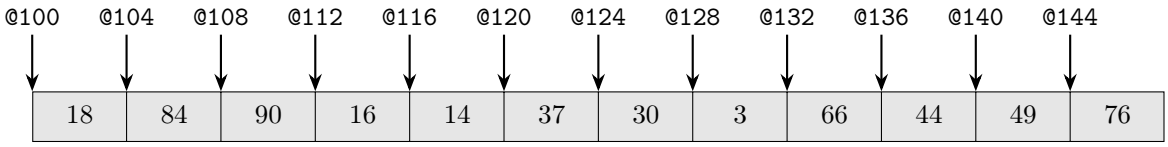


Figure 1.2: The possible layout of a C++ array `std::array<int, 12> L` in memory. Here, the notation `@100` represents the 100-th byte in memory. Hence, this array starts at memory address 100 and each `int` value takes up 4 bytes. The value `L[5]` (the value 37 in this example) can be found at address $100 + 4 \cdot 5 = 120$.

Many data structures represent *collections* of values on which a limited set of operations is allowed. Typically, such collections can be implemented in many ways, each with their own benefits and drawbacks.

Efficient algorithms require specific types of highly-efficient data structures. Hence, if we want to formally argue whether an algorithm is correct or efficient, we typically also have to be able to argue the correctness and efficiency of data structures.

In these notes, we only provide an introduction to the vast field of algorithms and data structures. In specific, we will focus on the following main topics:

1. the analysis of elementary iterative and recursive algorithms;
2. fundamental search algorithms and their applications: linear search and binary search;
3. elementary data collections: bags, stacks, and queues;
4. elementary data structures: arrays, singly linked lists, doubly linked lists, and dynamic arrays;
5. the practice and theory of sorting: merge sort, quick sort, counting sort, and radix sort;
6. tree-based data structures: trees, Huffman trees, heaps, search trees, and tries;
7. hash-based data structures; and
8. elementary graph representations and graph algorithms.

These main topics cover many essential topics that will come in handy during day-to-day software development. For example, these topics cover the majority of the standard data structures and algorithms provided by the standard libraries of popular programming language.

Remark 1.4. The notes take a particular focus on *data processing* and will take inspiration from algorithms used in the setting of databases. This is not a coincidence: many fundamental algorithms and data structures, including those part of standard programming languages, are designed to perform data processing tasks efficiently.

These notes are *not* a full replacement for a text book on algorithms and data structures: many excellent materials exist that provide an alternative or a more in-depth view of the material presented in these notes. We recommend students to explore these materials to aid in their current or future studies. Some examples:

1. Introduction to Algorithms, 4th edition, *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, and *Clifford Stein*. (2022). The MIT Press.

   This book can serve as an excellent all-round reference that covers all common fundamental algorithms and data structures, including all material from these notes.

2. Handbook of Data Structures and Applications, 2nd Edition, *Dinesh P. Mehta* and *Sartaj Sahni*. (2018). CRC Press.

   This book covers many advanced data structures (e.g., Fibonacci heaps) and can serve as an excellent starting point when exploring more advanced data structures.

3. Algorithms, 4th edition, *Robert Sedgewick* and *Kevin Wayne*. (2011). Addison-Wesley.

   This book covers most of the material from these notes with a more practical focus, e.g., by including details on how to implement the algorithms and data structures in the Java programming language.

4. Programming: The Derivation of Algorithms, *Anne Kaldewaij*. (1990). Prentice-Hall.

   This book provides an in-depth introduction to writing correct programs by deriving correct algorithms alongside their formal correctness proofs.

Chapter 2

Preliminaries

We expect that students can independently write and debug programs. In addition, the notes will heavily depend on a basic knowledge of *calculus* (e.g., elementary manipulation of arithmetic expressions involving sums, products, logarithms, and limits) and basic experience with *formal reasoning* (e.g., predicate logic, set notation, induction proofs, and proofs by contradiction). In this chapter, we will summarize this prior knowledge and detail the notations used in these notes.

2.1 Mathematics

Elementary notation

- $\lceil x \rceil$ is the value x, rounded up.
- $\lfloor x \rfloor$ is the value x, rounded down.
- $|x|$ is the absolute value of x.
- $[a, b)$ is the interval from a (inclusive) to b (exclusive): the values $a, a+1, ..., b-1$.
  An alternative notation that is used frequently is $[a, b[$.
- $[a, b]$ is the interval from a (inclusive) to b (inclusive): the values $a, a+1, ..., b$.
- We extend the above interval notations to lists and arrays: if L is a list with N values, then $L[a, b)$ is the list with values $L[a], \ldots, L[b-1]$. In these notes, lists typically start at zero, hence, we can write $L[0, N)$ to denote that a list has N values $L[0], \ldots, L[N-1]$.
- $\mathbb{N}$ are the natural numbers (non-negative whole numbers) and $\mathbb{Z}$ are the integers (the whole numbers).
- $f : A \to B$ identifies a function f with *domain* A and *range* (or *codomain*) B: the function f maps each input value from A to a unique value in B. The notation $f : A \to B : x \mapsto f(x)$ not only describes the domain and range of f, but also how f maps values from A to B. For example, $f : \mathbb{N} \to \mathbb{Z} : x \mapsto x^2$ is a function that takes as input non-negative whole numbers and maps each input x onto the whole number x^2.

  A function must be defined for all values of its domain, but does not have to use all values of its range (e.g., the above function uses only the positive values in $\mathbb{Z}$). If the domain and range are clear from the context, then we typically omit them: e.g., we typically simply write $f(x) = x^2$.

Divisions and fractions We write $a \operatorname{div} b$, with $a, b \in \mathbb{N}$ to denote $\left\lfloor \frac{a}{b} \right\rfloor$. We often write a/b to denote $\frac{a}{b}$.

- $x = \frac{xa}{a}$ for all $a \neq 0$.
- $\frac{a}{(\frac{b}{c})} = \frac{a \cdot c}{b}$.

- $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$.
- $\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$.

Powers and roots In these notes, we will only see roots $\sqrt[b]{c}$ for positive c.

- If $a^b = c$, then $\sqrt[b]{c} = a$.
- $\sqrt[b]{a^b} = a$.
- $n^a \cdot n^b = n^{a+b}$.
- $(n^a)^b = n^{a \cdot b}$.
- $n^{-a} = \frac{1}{n^a}$.
- $\sqrt[b]{n} = n^{\frac{1}{b}}$.

Logarithms In these notes, we will only see logarithms $\log_a(c)$ for positive bases a.

- If $a^b = c$, then $\log_a(c) = b$.
- $\log_a(a) = 1$ and $\log_a(1) = 0$.
- $a^{\log_a(n)} = n$.
- $\log_a(n \cdot m) = \log_a(n) + \log_a(m)$.
- $\log_a\left(\frac{n}{m}\right) = \log_a(n) - \log_a(m)$.
- $\log_a(n^b) = b \cdot \log_a(n)$.
- $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$ for all bases b.

Summations The summation $\sum_{i=a}^{b} f(i)$, with $f(i)$ an expression depending on i, represents the sum of terms

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \cdots + f(b).$$

For example, we can choose $a = 5$, $b = 8$, and $f(i) = (4+i)$ and we have $\sum_{i=5}^{8}(4+i) = (4+5)+(4+6)+(4+7)+(4+8)$.

- $\sum_{i=m}^{n} c = (n - m + 1) \cdot c$ with c a constant (an expression that does not use i).
- $\sum_{i=m}^{n}(f(i) + g(i)) = \left(\sum_{i=m}^{n} f(i)\right) + \left(\sum_{i=m}^{n} g(i)\right)$ with $f(i)$ and $g(i)$ expressions.
- $\sum_{i=m}^{n} c \cdot f(i) = c \cdot \left(\sum_{i=m}^{n} f(i)\right)$ with c a constant (an expression that does not use i) and $f(i)$ an expression.
- *Gauss:*
$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$
- *Geometric series:*
$$\sum_{i=0}^{n} ac^i = a\sum_{i=0}^{n} c^i = a\frac{c^{n+1} - 1}{c - 1}.$$
- *Geometric series, $c < 1$:*
$$\sum_{i=0}^{\infty} ac^i = \frac{a}{1 - c}.$$

- Special cases of the geometric series:

$$\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = 2^{n+1}-1;$$
$$\sum_{i=0}^{n} \left(\frac{1}{2}\right)^n = \frac{\left(\frac{1}{2}\right)^{n+1}-1}{\frac{1}{2}-1} = \frac{\left(\frac{1}{2}\right)^{n+1}-1}{-\frac{1}{2}} = -2\cdot\left(\left(\frac{1}{2}\right)^{n+1}-1\right) = 2-\left(\frac{1}{2}\right)^n;$$
$$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^n = \frac{1}{1-\frac{1}{2}} = \frac{1}{\frac{1}{2}} = 2.$$

Products Depending on the context, we write ab, $a\cdot b$, or $a\times b$ to denote "a multiplied by b". The product $\prod_{i=a}^{b} f(i)$, with $f(i)$ an expression depending on i, represents the product of terms

$$\prod_{i=a}^{b} f(i) = f(a)\times f(a+1)\times f(a+2)\times\cdots\times f(b).$$

For example, $\prod_{i=5}^{8}(4+i) = (4+5)\cdot(4+6)\cdot(4+7)\cdot(4+8)$.

- $\log\left(\prod_{i=a}^{b} f(i)\right) = \sum_{i=a}^{b}\log(f(i))$ with $f(i)$ an expression.
- *Factorial:*

$$a! = \prod_{i=1}^{a} i = (a)\cdot(a-1)\cdot(a-2)\cdot\cdots\cdot 2\cdot 1.$$

Limits *Knowledge of limits is not strictly required for these notes*, but can help with deriving some of the rule of thumbs introduced in Section 3.1.4. We *informally* say that $\lim_{n\to a} f(n) = L$ if the value $f(n)$ gets arbitrary close to L (we can make the distance $|L-f(n)|$ between L and $f(n)$ arbitrary small by choosing n sufficiently close to a). We *informally* say that $\lim_{n\to\infty} f(n) = L$ if the value $f(n)$ gets arbitrary close to L when we take values for n sufficiently large. We say that $\lim_{n\to a} f(n) = \infty$ if the value $f(n)$ becomes arbitrary large. Note that ∞ is *not a number* and cannot be used as such.

- $\lim_{n\to a}(f(n)+g(n)) = (\lim_{n\to a} f(n)) + (\lim_{n\to a} g(n))$.
- $\lim_{n\to a}(f(n)\cdot g(n)) = (\lim_{n\to a} f(n))\cdot(\lim_{n\to a} g(n))$.
- $\lim_{n\to a}(c\cdot f(n)) = c\cdot(\lim_{n\to a} f(n))$ for every constant c.
- $\lim_{n\to a}\left(\frac{f(n)}{g(n)}\right) = \frac{\lim_{n\to a} f(n)}{\lim_{n\to a} g(n)}$.
- $\lim_{n\to a}(f(n))^c = (\lim_{n\to a} f(n))^c$ for every constant $c>0$ if $(\lim_{n\to a} f(n)) > 0$.

2.2 Elementary logic and sets

Propositional logic We have the values *true* and *false* and the logical connectives *conjunction* or *and* ($\wedge$), *disjunction* or *or* ($\vee$), *negation* or *not* ($\neg$), *implication* ($\implies$), and *bi-implication* or *iff* ($\iff$). These logical connectives are defined according to the following truth table:

A	B	$A\wedge B$	$A\vee B$	$\neg A$	$A \implies B$	$A \iff B$
false	false	false	false	true	true	true
false	true	false	true	true	true	false
true	false	false	true	false	false	false
true	true	true	true	false	true	true

In spoken language, we often say that B is a *(logical) consequence* of A, B *if* A, or *if* A *then* B whenever $A \implies B$ is *true*. Likewise, we say A *if and only if* B whenever $A \iff B$ is *true*.

Formally, we write $f \equiv g$ to denote that logical expressions f and g are equivalent. We have:

- *Implication:* $A \implies B \equiv \neg A \vee B$.
- *Bi-implication:* $A \iff B \equiv (A \implies B) \wedge (B \implies A)$.
- *Double negation:* $\neg\neg A \equiv A$
- *De Morgan's laws:* $\neg(A \vee B) \equiv \neg A \wedge \neg B$ and $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- *Contraposition:* $A \implies B \equiv \neg B \implies \neg A$.

Predicate logic We write "$\forall x\ e(x)$" with $e(x)$ an expression dependent on x, to express that the expression $e(x)$ holds *for all* possible values of x. Likewise, we write "$\exists x\ e(x)$" to express that there *exists* a value of x for which the expression $e(x)$ holds. Examples:

$$\forall n\ (n \geq 0 \implies \sqrt{n^2} = n);$$
$$\exists n\ (n < 0 \wedge n^2 = 25).$$

In the above examples, we see two common patterns: $\forall x\ (D(x) \implies e'(x))$ and $\exists x\ (D(x) \wedge e'(x))$. In both cases, the expression $D(x)$ determines the *domain of the values* of x for which the expression $e'(x)$ is evaluated. One often encounters the following shorthand notation for these patterns:

$$\forall n \geq 0\ (\sqrt{n^2} = n);$$
$$\exists n < 0\ (n^2 = 25).$$

We have $\neg\forall x\ e(x) \equiv \exists x\ \neg e(x)$ and $\neg\exists x\ e(x) \equiv \forall x\ \neg e(x)$.

Sets A set is a collection of unique elements. Sets can have a finite number of elements (e.g., the set $\{apple, banana\}$) or an infinite number of elements (e.g., the whole numbers $\mathbb{Z}$). We use the notation

$$\{element \mid \text{conditions on } element\}$$

to specify a set in which every element satisfies the stated conditions. For example,

$$\{n \mid (n \in \mathbb{N}) \wedge (n \text{ is divisible by } 2)\}$$

to specify the set of all non-negative even numbers. We often abbreviate the above example as

$$\{n \in \mathbb{N} \mid n \text{ is divisible by } 2\}.$$

If S is a set and e is an element, then $e \in S$ *holds* if e is part of set S. We write $e \notin S$ as a shorthand for $\neg(e \in S)$. If S is a set with exactly n distinct elements, then we write $|S| = n$. We denote the empty set by $\emptyset$ (hence, $|\emptyset| = 0$). The fundamental set operations *union* ($\cup$), *intersection* ($\cap$), and *difference* ($\setminus$) are used to form new sets by combining sets:

- $V \cup W$. The set of all elements in either V or W: $\{e \mid (e \in V) \vee (e \in W)\}$.
- $V \cap W$. The set of all elements in both V and W: $\{e \mid (e \in V) \wedge (e \in W)\}$.
- $V \setminus W$. The set of all elements in V, but not in W: $\{e \in V \mid e \notin W\}$.

The standard operations to compare the content of sets are *subset* and *strict subset* ($\subseteq$ and $\subset$), *equality* and *inequality* ($=$ and $\neq$), and *superset* and *strict superset* ($\supseteq$ and $\supset$):

- $V \subseteq W$. Holds if every element in V is also in W: $\forall e \in V\ (e \in W)$.

- $V \supseteq W$. Holds if every element in W is also in V: $W \subseteq V$.
- $V = W$. Holds if V and W hold the same elements: $(V \subseteq W) \wedge (W \subseteq V)$.
- $V \neq W$. Holds if V and W do not hold the same elements: $\neg(V = W)$.
- $V \subset W$. Holds if every element in V is also in W, while some elements of W are not in V: $(V \subseteq W) \wedge (V \neq W)$.
- $V \supset W$. Holds if every element in W is also in V, while some elements of V are not in W: $W \subset V$.

2.3 Proofs

We expect students to be familiar with standard proof techniques such as *counterexamples*, *proof by induction* and *proof by contradiction*. Next, we provide examples of each of these proof techniques. We expect *familiarity* with these proof techniques: you should be able to write the skeleton of a proof without effort: in these notes, your effort should be directed toward the hard parts of a proof.

In general, proving a statement is *hard* and requires a great deal of creativity. Well-written proofs hide this complexity very well, however: in a well-written proof, every proof step is logical and makes sense. That makes carefully-written proofs deceivingly easy-to-follow (we did our best to ensure this also applies to the proofs in these notes). Reading and fully understanding easy-to-follow proofs does *not* train in *writing your own proofs*, however! We recommend using the examples in these notes mainly as inspiration toward the kinds of techniques you can utilize, and not as training in itself.

Remark 2.1. Proofs cannot magically create facts out of thin air! Any fact derived during a proof step should follow from the facts known prior to that point. In practice, many proofs are for claims of the form "if *conditions*, then *fact of interest*". In such a proof, you can assume the facts given by *conditions*. Typically, proofs also rely on relevant prior knowledge, e.g., in a proof involving numbers one often relies on standard mathematical facts such as those presented in Section 2.1.

2.3.1 Counterexample

To prove that a claim *does not hold*, you only have to show a single instance in which the claim is false.

For example, to show that $2^n < 100n$ does not hold, one can simply fill in $n = 10$ for which $2^n = 1024$ and $100n = 1000$. As $1024 < 1000$ does not hold, $2^n < 100n$ does not hold.

2.3.2 Proof by induction

A proof by induction for a claim $C(n)$ has the following basic form:

- determine the *base cases* of claim $C(n)$, e.g., the base cases are $C(0)$ and $C(1)$;
- *prove the base cases*, e.g., prove that the claims $C(0)$ and $C(1)$ hold;
- formulate an *induction hypothesis* based on claim $C(n)$: an induction hypothesis is always of the form

  "assume that your claim $C(n)$ holds for all values of i up-to-some value bounded by m",

  e.g., we assume that $C(i)$ holds for all n, $0 \leq n < m$.
- prove the *inductive step*: assume that the induction hypothesis holds, and use this fact to prove that your claim holds for the next possible value, e.g., prove that $C(m)$ holds. In this proof, you eventually want to reduce the proof for claim $C(m)$ to a preceding claim $C(n)$, $n < m$, such that the induction hypothesis can be applied.

Alternatively, induction hypotheses can be of the form "we assume that $C(n)$ holds for all $0 \leq n \leq m$". If we use such an induction hypothesis, then the inductive step needs to prove the case $C(m+1)$. As errors with additional terms "$+1$" are easily made, claims of the form $C(m)$ are typically slightly easier to prove than claims of the form $C(m+1)$, however.

As a first example, we will prove a special case of the geometric series shown in Section 2.1 using induction:

Theorem 2.2. *We have* $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Proof. We prove this theorem using induction. Our base case is $n = 0$, in which case we have $\sum_{i=0}^{0} 2^i = 2^0 = 1$ and we have $2^{0+1} - 1 = 1$. Hence, $\sum_{i=0}^{0} 2^i = 2^{0+1} - 1$ holds.

As the induction hypothesis, we assume that $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ holds for all $0 \leq n < m$.

Next, we prove that $\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$ holds. We have

$$\sum_{i=0}^{m} 2^i = 2^m + \sum_{i=0}^{m-1} 2^i.$$

As $m - 1 < m$, we can apply the induction hypothesis on $\sum_{i=0}^{m-1} 2^i$ to obtain

$$\sum_{i=0}^{m} 2^i = 2^m + 2^{m-1+1} - 1 = 2^m + 2^m - 1.$$

As $2^m + 2^m = 2 \cdot 2^m$ and $2 \cdot 2^m = 2^{m+1}$, we conclude

$$\sum_{i=0}^{m} 2^i = 2^m + 2^m - 1 = 2^{m+1} - 1.$$ □

As a second example, we will prove the formula of Gauss shown in Section 2.1 using induction:

Theorem 2.3. *We have* $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Proof. We prove this theorem using induction. Our base case is $n = 0$, in which case we have $\sum_{i=0}^{0} i = 0$ and $\frac{0(0+1)}{2} = 0$. Hence, $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$ holds.

As the induction hypothesis, we assume that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ holds for all $0 \leq n < m$.

Next, we prove that $\sum_{i=0}^{m} i = \frac{m(m+1)}{2}$ holds. We have

$$\sum_{i=0}^{m} i = m + \sum_{i=0}^{m-1} i.$$

As $m - 1 < m$, we can apply the induction hypothesis on $\sum_{i=0}^{m-1} i$ to obtain

$$\sum_{i=0}^{m} i = m + \frac{(m-1)(m-1+1)}{2} = m + \frac{m(m-1)}{2}.$$

As $m = \frac{2m}{2}$ and $m(m-1) + 2m = m(m+1)$, we conclude

$$\sum_{i=0}^{m} i = m + \frac{m(m-1)}{2} = \frac{m(m+1)}{2}.$$ □

The complexity of many induction proofs hides in the choice of the induction hypothesis. Typically, completing a proof by induction requires systematic trial and error: you make a best-effort guess $G(i)$ for the induction hypothesis, work out the proof using this guess $G(i)$, and when you run into issues you *refine or replace* guess $G(i)$ by a better induction hypothesis.

Remark 2.4. On the one hand, induction is an extremely powerful tool. On the other hand, induction is a blunt-force tool that does not always provide much *insight*. To illustrate this, we next provide an *alternative* proof for Theorem 2.3.

Proof. Consider the sum $\sum_{i=0}^{n} i$. We have

$$\sum_{i=0}^{n} i = 0 + 1 + \cdots + (n-1) + n.$$

We can multiply each side by two to obtain

$$2 \cdot \left(\sum_{i=0}^{n} i \right) = \Big(0 + 1 + \cdots + (n-1) + n\Big) + \Big(0 + 1 + \cdots + (n-1) + n\Big).$$

As addition is *commutative*, we can reorder the terms in $0+1+\cdots+(n-1)+n$ to obtain $n+(n-1)+\cdots+1+0$:

$$2 \cdot \left(\sum_{i=0}^{n} i \right) = \Big(0 + 1 + \cdots + (n-1) + n\Big) + \Big(n + (n-1) + \cdots + 1 + 0\Big).$$

We can further reorder the sum on the right by grouping the i-th term, $0 \leq i \leq n$, in $(0+1+\cdots+(n-1)+n)$ with the i-th term in $(n+(n-1)+\cdots+1+0)$. We obtain

$$2 \cdot \left(\sum_{i=0}^{n} i \right) = \Big((0+n) + (1+(n-1)) + \cdots + ((n-1)+1) + (n+0)\Big).$$

In the sum on the right, each grouped term is equivalent to n and we have $n+1$ such groups. Hence,

$$2 \cdot \left(\sum_{i=0}^{n} i \right) = (n+1) \cdot n$$

and we conclude

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

□

2.3.3 Proof by contradiction

A proof by contradiction for a claim C has the following basic form.

- Assume that claim C *does not hold*.
- Use this assumption to prove a *contradiction*: if claim C does not hold, then we can prove that both statement S and statement $\neg S$ hold (for some statement S).
- As we proved a contradiction, our original assumption that C does *not hold* is *not a valid assumption*. Hence, by contradiction, we have proven that C holds.

As an example, we prove the following:

Theorem 2.5. *The number $\sqrt{2}$ is not a rational number (a number that can be written as a fraction of two integers).*

Proof. We prove this theorem using contradiction. We assume $\sqrt{2}$ is a rational number. Hence, there exists two integers a and b such that $\sqrt{2} = a/b$. Without loss of generality, we can assume that a and b do *not* have a common divisor c, as otherwise we can simplify the fraction by dividing both a and b by their common divisors. Hence, without loss of generality, we assume that the fraction a/b **is an *irreducible fraction***. Next, we square the term $\sqrt{2} = a/b$ and obtain $2 = a^2/b^2$. As $2 = a^2/b^2$, we have $2b^2 = a^2$ and we conclude that a^2 is even (can be divided by two). Consequently, also a is even. Let $a = 2 \cdot a'$ for some integer a'. We have $a^2 = 4a'^2$, $2b^2 = 4a'^2$, and $b^2 = 2a'^2$. Now, we can conclude that b^2 is even, due to which also b is even. As both a and b are even, they have a common divisor $c = 2 > 1$. Hence, a/b **is *not* an irreducible fraction**, a contradiction. By contradiction, we conclude that $\sqrt{2}$ cannot be a rational number. □

Chapter 3

The Analysis of Algorithms

In this chapter, we introduce a formal framework that will allow us to *analyze algorithms* and formally answer questions about the correctness and efficiency of algorithms. We develop this framework by example while studying a few simple yet essential algorithms.

3.1 The Contains Algorithm

The first algorithm we look at is an algorithm to check whether a value is *contained* in a list. Remember from Definition 1.1 that an algorithm is a description on how to solve a specific problem. Hence, we first define the *contains problem*:

Problem 3.1. Let L be a list and v be a value. A solution to the *contains problem* for L and v will compute `true` ('found') if the value v is in list L and will compute `false` ('not found') otherwise.

Next, we write the CONTAINS algorithm of Figure 3.1 to solve Problem 3.1.

Algorithm CONTAINS(L, v) **:**

```
1:  i, r := 0, false.
2:  while i ≠ |L| do
3:    if L[i] = v then
4:      r := true.
5:      i := i + 1.
6:    else
7:      i := i + 1.
8:    end if
9:  end while
10: return r.
```

Figure 3.1: The CONTAINS algorithm.

Remark 3.2. The CONTAINS algorithm of Figure 3.1 is not very elegant: most people with programming experience are able to write a better and more elegant version. The version we present here is easier to analyze than more elegant versions, however. Hence, as this is our first attempt at analyzing an algorithm, we will use our non-elegant version.

We can now ask whether the CONTAINS algorithms is correct and efficient.

3.1.1 Correctness of CONTAINS

To formally determine whether any algorithm is correct, we first have to determine *what* the algorithm is supposed to achieve. Hence, the first step in proving the correctness of CONTAINS is *specifying* what the

result should be. In this case, CONTAINS should solve the contains problem of Problem 3.1. Hence, CONTAINS should return `true` if $v \in L$ and `false` otherwise. Such a predicate on the result of an algorithm is typically referred to as the *post-condition* of CONTAINS. Typically, algorithms *cannot* operate on any arbitrary input. For example, CONTAINS operates on an *array* L and a value v. We call such a predicate that expresses the initial conditions the *pre-conditions* of CONTAINS. In Figure 3.2, we have modified the CONTAINS algorithm by including the pre-condition and post-condition.

Algorithm CONTAINS(L, v) **:**
Pre: L is an *array*, v a value.
1: $i, r := 0,$ `false`.
2: **while** $i \neq |L|$ **do**
3: **if** $L[i] = v$ **then**
4: $r :=$ `true`.
5: $i := i + 1$.
6: **else**
7: $i := i + 1$.
8: **end if**
9: **end while**
10: **return** r.
Post: return `true` if $v \in L$ and `false` otherwise.

Figure 3.2: The CONTAINS algorithm, now with pre-conditions and post-conditions.

We start our first correctness proof by specifying the *goal* of the proof: we assume that the pre-condition holds *before* execution of Line 1, and we have to prove that execution of CONTAINS ends such that the post-condition holds *after* execution of Line 10.

As our first proof steps, we look at the statements at Line 1 and Line 10. We have assumed that the pre-condition holds *before* execution of Line 1. Hence, we can derive what *should hold* after execution of Line 1. Execution of Line 1 performs a simple assignment that initializes new variables i and r and after execution, we know that $i = 0$ and $r =$ `false`. In addition, the pre-condition still holds as we have not changed L or v.

We have not yet determined what holds *before* execution of Line 10, but we do know that after execution of this line the post-condition should hold. Hence, we can derive what *should hold* before execution of Line 10. Execution of Line 10 returns the value r. According to the post-condition, the returned value should be `true` if $v \in L$ and `false` otherwise. Hence, *before* execution of Line 10, we must have "$r =$ `true` if $v \in L$ and $r =$ `false` otherwise".

Our remaining proof goal is as follows: we assume that the predicate "pre-condition, $i = 0$, and $r =$ `false`" holds before execution of the while loop at Line 2, and we have to prove that the predicate "$r =$ `true` if $v \in L$ and $r =$ `false` otherwise" holds after execution of the while loop (after Line 9). We have summarized this goal in Figure 3.3.

The while loop can be *repeated* many times. After each repetition of the loop of Line 2, different predicates will hold on the state of the program. For example, the value of i will change each repetition of the while loop and the value of r can change.

We note that repetition, e.g., in the form of a while loop such as at Line 2 or in the form of recursion, is at the center of most non-trivial algorithms. If we want to prove that a predicate holds *after* such repetition, then that proof needs to be able to argue over an arbitrary number of repeated statements. In our overview of proof methods in Section 2.3, we have only seen one proof technique that seems suitable for reasoning about *repeated* steps: *induction*. Hence, we will use induction to reason about the while loop.

We use the basic form of inductive proofs outlined in Section 2.3.2. We claim that the predicate

$$C(i) = \text{"}0 \leq i \leq |L|,\ v \in L[0, i) \text{ implies } r = \texttt{true}, \text{ and } v \notin L[0, i) \text{ implies } r = \texttt{false}.\text{"}$$

always holds when starting execution of Line 2. As we are using induction, we have to prove a base case, formalize an induction hypothesis, and prove the inductive step:

Algorithm CONTAINS(L, v) **:**
Pre: L is an *array*, v a value.
1: $i, r := 0,$ `false`.
   /* L is an *array*, v a value, $i = 0$, and $r =$ `false` */
2: **while** $i \neq |L|$ **do**
3: **if** $L[i] = v$ **then**
4: $r :=$ `true`.
5: $i := i + 1$.
6: **else**
7: $i := i + 1$.
8: **end if**
9: **end while**
   /* $r =$ `true` if $v \in L$ and $r =$ `false` otherwise */
10: **return** r.
Post: return `true` if $v \in L$ and `false` otherwise.

Figure 3.3: The CONTAINS algorithm, now with pre-conditions, post-conditions, and the predicates that should hold *before* and *after* the while loop.

- *Base case*. The base case proves that our claim holds the *first time* we reach Line 2. The first time we reach Line 2 is right after execution of Line 1. At that point, we have already determined that the predicate "L is an *array*, v a value, $i = 0$, and $r =$ `false`" holds. We have $r =$ `false` and we note that the array $L[0, i) = L[0, 0)$ is empty and, hence, does not hold any values. As such, we have $0 \leq i \leq |L|$, $v \notin L[0, i)$ and $r =$ `false`, and we conclude $C(0)$ holds.
- *Induction hypothesis*. After the j-th, $j < m$, repetition of the while loop, the predicate $C(i)$ holds.
- *Inductive step*. The goal of the inductive step is to prove that the while loop maintains the induction hypothesis: *if* the induction hypothesis holds when we start execution of the body of the while loop at Line 2, then after execution of Lines 2–8 (the body of the while loop) the inductive hypothesis must hold again.

  The first step of the while loop, at Line 2, checks whether $i \neq |L|$. Hence, when we start execution of Line 3, we know that $i \neq |L|$. As we have not yet made any changes, the induction hypothesis still holds. By the induction hypothesis, we also know $0 \leq i \leq |L|$ and, combined with $i \neq |L|$, we have $0 \leq i < |L|$. Next, we check the condition $L[i] = v$ by executing Line 3. Based on whether the condition $L[i] = v$ holds, we distinguish two cases:

  1. *The condition $L[i] = v$ holds*. The algorithm first executes Line 4 and then executes Line 5. Let i_{old} and i_{new} be the values of i *before* and *after* executing these two lines.

     Before the algorithm executes Line 4, we know that $0 \leq i_{\text{old}} < |L|$ and we know $L[i_{\text{old}}] = v$ due to Line 3. After execution of Line 5, we additionally have $r =$ `true` due to Line 4 and $i_{\text{new}} = i_{\text{old}} + 1$ due to Line 5. Using these facts, we need to prove that the induction hypothesis holds again: we need to prove that the predicate $C(i_{\text{new}})$ holds.

     From $L[i_{\text{old}}] = v$, we can derive $v \in L[0, i_{\text{old}} + 1)$. As $i_{\text{old}} + 1 = i_{\text{new}}$, we can also derive $v \in L[0, i_{\text{new}})$. Finally, from $0 \leq i_{\text{old}} < |L|$ and $i_{\text{old}} + 1 = i_{\text{new}}$, we can derive $0 \leq i_{\text{new}} \leq |L|$. Hence, as $0 \leq i_{\text{new}} \leq |L|$, $v \in L[0, i_{\text{new}})$, and $r =$ ' found', we conclude $C(i_{\text{new}})$ holds
  2. *The condition $L[i] = v$ does not hold*. The algorithm only executes Line 7. Let i_{old} and i_{new} be the values of i *before* and *after* executing this line.

     Before the algorithm executes Line 7, we know that $0 \leq i_{\text{old}} < |L|$, $L[i_{\text{old}}] \neq v$ due to Line 3, and $C(i_{\text{old}})$ holds due to the induction hypothesis. We note that only the value of i changes: due to Line 7, we know $i_{\text{old}} + 1 = i_{\text{new}}$. Hence, from $0 \leq i_{\text{old}} < |L|$ and $i_{\text{old}} + 1 = i_{\text{new}}$, we can derive $0 \leq i_{\text{new}} \leq |L|$. Based on the value of r, we distinguish two cases:

(a) $r = \texttt{true}$. By $C(i_{\text{old}})$, we have $v \in L[0, i_{\text{old}})$. As such, we have $v \in L[0, i_{\text{old}} + 1)$ and, hence, $v \in L[0, i_{\text{new}})$. We conclude $C(i_{\text{new}})$ holds.

(b) $r = \texttt{false}$. By $C(i_{\text{old}})$, we have $v \notin L[0, i_{\text{old}})$. By $L[i_{\text{old}}] \neq v$, we also have $v \notin L[0, i_{\text{old}} + 1)$ and, hence, $v \notin L[0, i_{\text{new}})$. We conclude $C(i_{\text{new}})$ holds.

By induction, we conclude that *after* execution of the while loop, the predicate $C(i)$ holds. The while loop of Line 2 only ends if $i \neq |L|$ no longer holds. Hence, after execution of the while loop, the predicate $\neg(i \neq |L|)$ also holds. We conclude that after Line 9, we have "$C(i)$ and $\neg(i \neq |L|)$".

What remains to prove is that the predicate "$C(i)$ and $\neg(i \neq |L|)$" that holds after Line 9 implies the predicate "$r = \texttt{true}$ if $v \in L$ and $r = \texttt{false}$ otherwise" that must hold before execution of Line 10. By $\neg(i \neq |L|)$, we conclude $i = |L|$. Hence, $C(|L|)$ holds and we have "$v \in L[0, |L|)$ implies $r = \texttt{true}$ and $v \notin L[0, |L|)$ implies $r = \texttt{false}$". As $L[0, |L|) = L$, the predicate "$r = \texttt{true}$ if $v \in L$ and $r = \texttt{false}$ otherwise" that must hold before execution of Line 10 holds.

⋆ *Remark* 3.3. You might wonder why the claim $C(i)$ includes the predicate $0 \leq i \leq |L|$: we do not use this predicate after the while loop. During the while loop, we do need the fact that i is a valid array index, however: e.g., otherwise reading $L[i]$ at Line 3 would not make sense. The guard $i \neq |L|$ of the while loop at Line 2 strictly speaking does *not* guarantee this: $i \neq |L|$ not only holds for valid array indices, but also for $i < 0$ or $i > |L|$. Hence, our proof needs to be able to establish that within the while loop, the variable i is a valid array index. The predicate $0 \leq i \leq |L|$ together with the guard $i \neq |L|$ allow us to do so.

In correctness proofs, the induction hypothesis used to prove the correctness of a loop is typically called an *invariant*. Using this terminology, we have summarized the above proof steps in Figure 3.4.

Algorithm CONTAINS(L, v) **:**

Pre: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.
   /* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$ */
   /* invariant: $C(i) =$ "$0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, and $v \notin L[0, i)$ implies $r = \texttt{false}$" */
2: **while** $i \neq |L|$ **do**
      /* $C(i)$ (invariant) and $i \neq |L|$ */
3:    **if** $L[i] = v$ **then**
         /* $C(i)$ (invariant), $i \neq |L|$, and $L[i] = v$ */
4:       $r := \texttt{true}$.
         /* $i \neq |L|$, $L[i] = v$, and $r = \texttt{true}$ */
5:       $i := i + 1$.
         /* $C(i_{\text{old}})$ (invariant with i replaced by i_{old}), $i_{\text{old}} < |L|$, $L[i_{\text{old}}] = v$, $r = \texttt{true}$, and $i_{\text{new}} = i_{\text{old}} + 1$ */
6:    **else**
         /* $C(i)$ (invariant), $i \neq |L|$, and $L[i] \neq v$ */
7:       $i := i + 1$.
         /* $C(i_{\text{old}})$ (invariant with i replaced by i_{old}), $i_{\text{old}} < |L|$, $L[i_{\text{old}}] \neq v$, and $i_{\text{new}} = i_{\text{old}} + 1$ */
8:    **end if**
      /* invariant */
9: **end while**
   /* $C(i)$ (invariant) and $\neg(i \neq |L|)$ */
   /* $r = \texttt{true}$ if $v \in L$ and $r = \texttt{false}$ otherwise */
10: **return** r.

Post: return $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise.

Figure 3.4: The CONTAINS algorithm, now with pre-conditions, post-conditions, invariants, and the predicates that should hold at every step.

At this point, it is a good idea to take a step back and ask whether we are finished with our proof: *did* we prove that the CONTAINS algorithm is correct? We did *certainly* prove that the post-condition holds whenever the algorithm finishes. We overlooked one small detail, however: we never proved that the algorithm will ever finish: it is easy to make mistakes in an algorithm and write an *infinite loop* that never finishes.

Hence, as the last step of our proof, we will have to show that the while loop terminates. To prove that a while loop terminates, we typically show that we can define a *bound function* on the state of the algorithm that satisfies the following two conditions:

- the output of the bound function is a natural number (0, 1, 2, ...);
- the output of the bound function strictly decreases after each iteration of the *loop body*.

If these two conditions hold on the bound function, then they imply that the loop-body can only be executed a bounded number of times: the output of the bound function can only strictly decrease a bounded number of times before it hits zero.

In this case, we can use the bound function $|L| - i$. The output of the function $|L| - i$ is a natural number. The value $|L| - i$ strictly decreases every iteration of the while loop, as the value i strictly increases due to Lines 5 and 7. Finally, due to the condition $i \neq |L|$ at Line 2, the while loop terminates when $|L| - i = 0$.

We can finally conclude that the CONTAINS algorithm is correct.

Theorem 3.4. *The CONTAINS algorithm of Figure 3.1 is correct and solves Problem 3.1.*

⋆ *Remark* 3.5. Technically, the above correctness proof sweeps some details under the rug. Consider the EVILCONTAINS algorithm of Figure 3.5. This algorithm has the same pre-condition and post-condition as CONTAINS but, clearly, does *not* solve the contains problem of Problem 3.1.

Algorithm EVILCONTAINS(L, v) **:**

Pre: L is an *array*, v a value.

1: $L := []$.
2: **return** `false`.

Post: return `true` if $v \in L$ and `false` otherwise.

Figure 3.5: The EVILCONTAINS algorithm.

Hence, technically EVILCONTAINS is a solution to the contains problem of Problem 3.1. The technicality used by EVILCONTAINS is simple to see: EVILCONTAINS changes the input array L and we have *never* specified in Problem 3.1 that one is not allowed to change L.

We can *fix* Problem 3.1 by simply specifying that the problem should be solved with respect to the original inputs L and v and that the algorithm is *not allowed* to change the inputs. With this fix, CONTAINS still solves the find problem, whereas EVILCONTAINS does not.

To not overburden the remainder of these notes with technicalities, we shall often simply assume that algorithms "*do not change their inputs*" *unless* clearly required by the problem being solved.

Exercise 3.1. We wrote the CONTAINS algorithm in such a way to simplify the proof of correctness. Our version is not very typical, however. Indeed, an experienced programmer is much more likely to write the CONTAINSFAST algorithm of Figure 3.6 instead.

Algorithm CONTAINSFAST(L, v) **:**

1: $i := 0$.
2: **while** $i \neq |L|$ **do**
3: **if** $L[i] = v$ **then**
4: **return** `true`.
5: **end if**
6: $i := i + 1$.
7: **end while**
8: **return** `false`.

Figure 3.6: The CONTAINSFAST algorithm.

Is the CONTAINSFAST algorithm correct? Can we reuse the correctness proof for CONTAINS?

3.1.2 ⋆ Formalizing proof obligations

In the previous section, we provided a formal proof of the correctness of CONTAINS. We did so by showing that whenever the pre-condition holds, then execution of CONTAINS results in the post-condition. The approach taken to prove the correctness of CONTAINS can be fully formalized by formalizing the proof obligations that are imposed by each type of statement S one can have in an algorithm. Next, we shall introduce such a formalization for a minimal set of statement types.

In a correctness proof, any predicate Q that holds after execution of a statement S stems from the predicate P that holds *right before* executing S and the operations performed by S.

Definition 3.6. A *Hoare triple* is a triple `/* P */ S /* Q */` that expresses the predicates P that hold *before* execution of statements S and the predicates Q that should hold *after* execution of statements S. A Hoare triple is *valid* if one can prove that predicates Q holds *after* execution of statements S, assuming that predicates P held *before* execution of statements S.

In this terminology, proving the correctness of a program reduces to proving that the program terminates and that the triple `/*` pre-condition `*/` *program body* `/*` post-conditions `*/` is a valid Hoare triple.

Next, we formalize the proof obligations for Hoare triples of the basic building blocks in an algorithm: assignment statements, concatenations of two statements, if-statements, and while loop statements. These four types of statements cover the main types of statements used in algorithms.

Remark 3.7. For brevity, we will only formalize these four types of statements. Throughout these notes, we will use several alternative types of statements, e.g., the for loop statement. Close inspection of these alternative statements will show that each of them is a shorthand for a construction that can be expressed in terms of the four types of statements we formalize here. We leave it as an exercise to the reader to determine the proof obligations for such shorthand notations.

The assignment. An assignment statement has the following form:

`/*` P `*/`
1: $v_1, \ldots, v_n := e_1, \ldots, e_n$.
`/*` Q `*/`.

Remark 3.8. In these notes, the assignment is a multi-assignment. First, all expressions $e_1, \ldots, e_n$ are evaluated to their resulting values, after which these resulting values are assigned to $v_1, \ldots, v_n$. Hence, the assignment $v, w := w, v$ will properly *swap* the values of v and w.

Let $v_{1,\text{old}}, \ldots, v_{n,\text{old}}$ and $v_{1,\text{new}}, \ldots, v_{n,\text{new}}$ be the values of $v_1, \ldots, v_n$ *before* and *after* execution of the assignment. After execution of the assignment, the following predicates hold:

1. The predicate P_{old} obtained from P by replacing each usage of $v_1, \ldots, v_n$ by $v_{1,\text{old}}, \ldots, v_{n,\text{old}}$.
2. The predicates A_i of the form "$v_{i,\text{new}} = e_{i,\text{old}}$", $1 \leq i \leq n$, in which $e_{i,\text{old}}$ is obtained from e_i by replacing each usage of $v_1, \ldots, v_n$ by $v_{1,\text{old}}, \ldots, v_{n,\text{old}}$.

Let Q_{new} be the predicate obtained from Q by replacing each usage of $v_1, \ldots, v_n$ by $v_{1,\text{new}}, \ldots, v_{n,\text{new}}$. To prove that the Hoare triple is valid, we need to prove that $(P_{\text{old}} \wedge A_1 \wedge \cdots \wedge A_n) \implies Q_{\text{new}}$ holds.

Example 3.9. Consider the following Hoare triple

`/*` P: $v > 10, w < 20$ `*/`
1: $v, w := 2 \cdot w, 3 \cdot v$.
`/*` Q: $w > 30, v < 40$ `*/`

After the assignment, the predicates

$$P_{\text{old}} = \text{“}v_{\text{old}} > 10, w_{\text{old}} < 20\text{”}; \qquad A_1 = \text{“}v_{\text{new}} = 2 \cdot w_{\text{old}}\text{”}; \qquad A_2 = \text{“}w_{\text{new}} = 3 \cdot v_{\text{old}}\text{”},$$

hold. To prove that the Hoare triple is valid, we must prove that $Q_{\text{new}} = \text{“}w_{\text{new}} > 30, v_{\text{new}} < 40\text{”}$ also holds. By $v_{\text{old}} > 10$ and $w_{\text{new}} = 3 \cdot v_{\text{old}}$, we conclude $w_{\text{new}} > 30$. By $w_{\text{old}} < 20$ and $v_{\text{new}} = 2 \cdot w_{\text{old}}$, we conclude $v_{\text{new}} < 40$. Hence, Q_{new} holds.

The concatenation of two statements. A concatenation has the following form:

```
   /* P */
1: first statement.
   /* ? */
2: second statement.
   /* Q */
```

To prove that the Hoare triple /* P */ *first statement; second statement* /* Q */ is valid, one has to find a predicate R such that the Hoare triples /* P */ *first statement* /* R */ and /* R */ *second statement* /* Q */ are both valid Hoare triples.

Example 3.10. Consider the following Hoare triple

```
   /* P: m, n are two integers */
1: v := m · n.
   /* R */
2: w := v · v.
   /* Q: w = m^2 · n^2 */
```

To prove that the above concatenation forms a valid Hoare triple, we choose the predicate $R =$ "$v = m \cdot n$". Next, one can prove that /* P */ $v := m \cdot n$ /* R */ and /* R */ $w := v \cdot v$ /* Q */ are valid Hoare triples by following the rules for proving an assignment.

The if-statement. An if-statement has the following form:

```
   /* P */
1: if condition then
      /* P and condition */
2:    if body.
      /* Q */
3: else
      /* P and ¬condition */
4:    else body.
      /* Q */
5: end if
   /* Q */
```

To prove that the Hoare triple /* P */ *if statement* /* Q */ is valid, we need to prove that

1. /* P and *condition* */ *if body* /* Q */ is a valid Hoare triple; and
2. /* P and ¬*condition* */ *else body* /* Q */ is a valid Hoare triple.

By doing so, we prove that /* Q */ holds after execution of the if-statement via a simple case distinction based on the condition *condition*.

Example 3.11. Consider the following Hoare triple

```
   /* P: m, n are two integers */
1: if m > n then
2:    v := m.
3: else
4:    v := n.
5: end if
   /* Q: v = max(m, n) */
```

To prove that this Hoare triple is valid, we must prove that

1. /* P and $m > n$ */ $v := m$ /* Q */ is a valid Hoare triple. After execution of $v := m$, we have $v = m$ and $m > n$. As $m > n$, $m = \max(m, n)$. As $v = m$, we conclude $v = \max(m, n)$.

2. /* P and $\neg(m > n)$ */ $v := n$ /* Q */ is a valid Hoare triple. After execution of $v := n$, we have $v = n$ and $\neg(m > n)$. As $\neg(m > n)$, we have $m \leq n$. Hence, $n = \max(m, n)$. As $v = n$, we conclude $v = \max(m, n)$.

The while-statement. A while-statement has the following form:

```
   /* P */
   /* invariant: I, bound function: f */
1: while guard do
      /* I and guard */
2:    loop body.
      /* I */
3: end while
   /* I and ¬guard */
   /* Q */
```

In Section 3.1.1, we already provided a detailed description of the proof obligations for a while-statement. Here, we only provide a summary. To prove that the Hoare triple /* P */ *while statement* /* Q */ is valid, we need to prove two properties of the statement:

1. First, we need to prove that the while loop maintains an invariant I:
   (a) Initially, invariant I must hold: we need to prove $P \implies I$.
   (b) After execution of the loop body, the invariant I must hold (the loop body maintains invariant I): we need to prove that /* I and *guard* */ *loop body* /* I */ is a valid Hoare triple.
   (c) After termination of the while loop, Q must hold: we need to prove $(I \wedge \neg guard) \implies Q$.
2. Second, we need to prove that the while loop terminates by providing a valid bound function f.

Finding an invariant typically starts with an *educated guess* based on the information already available, e.g., one can try to first derive the predicates that hold right before the while loop (working top-down) and the predicates one needs right after the while loop (working bottom-up), this based on the remainder of the program.

Remark 3.12. In practice, finding the right invariants to make a correctness proof work is the most challenging part of the proof. In complex programs, you often end up with an initial guess for an invariant that does not provide all the information necessary during your proof. In such cases, you can always try to *strengthen* the invariant by adding additional predicates to it.

Exercise 3.2. Prove that the ArraySum algorithm of Figure 1.1 is correct. What are the pre-conditions, post-conditions, invariant, and bound function?

Exercise 3.3. Consider the FastPower algorithm of Figure 3.7. Why does the pre-condition introduce terms X and Y (that are not used in the program itself)? Is the algorithm correct? If so, prove that the FastPower algorithm is correct. Otherwise, show how to fix the algorithm.

3.1.3 Efficiency of Contains

Next, we will look at the efficiency of the Contains algorithm. When studying the efficiency of an algorithm, we are mainly interested in the *scalability* of that algorithm: how do the costs (e.g., runtime or memory consumption) of the algorithm increase when increasing the size of the input. Next, we will focus on studying the runtime cost of the Contains algorithm. To do so, we want to make a *scientific model* of the runtime of Contains that allows us to easily make accurate predictions on the runtime.

Example 3.13. Assume that the runtime of Contains on my laptop with a list L with $N = |L| = 1000$ values is 12 µs. We want to be able to *predict* the runtime of Contains when we increase the size of list L, e.g., what is the runtime of Contains if we double the size of list L? What if we triple the size of list L?

Algorithm FastPower(x, y) **:**
Pre: $y \in \mathbb{N}$, $X = x$, and $Y = y$.
1: $r, t, n := 1, x, y$.
2: **while** $n \neq 0$ **do**
3: **if** n is even **then**
4: $t, n := t \cdot t, n/2$.
5: **else**
6: $r, n := r \cdot t, n - 1$.
7: **end if**
8: **end while**
9: **return** r.
Post: returns X^Y.

Figure 3.7: The FastPower algorithm that computes x^y.

The Contains algorithm itself is a bad model for its runtime: this model does not enable us to make predictions without first implementing Contains and measuring its performance. As a first attempt toward a *better* model, we can try to count how many *instructions* are performed by a processor that is executing the algorithm. Counting instructions is easier said than done, however: the exact count will depend on the programming language in which we implement Contains, the compiler that turned the source code into an executable, the settings of the compiler, the type of processor on which we will run the executable, and so on.

Next, we will perform a best-effort estimation. At Line 1, we perform two assignment operations (2 instructions). At Line 2, we first perform the comparison $i \neq N$ and then make a decision to either enter the loop or jump to Line 10 (2 instructions). At Line 3, we first read the value $L[i]$, then perform the comparison $L[i] = v$, and then make a decision to either execute Line 4 or Line 7 (3 instructions). At Line 4, we perform a single assignment (1 instruction). At Lines 5 and 7, we first compute $i + 1$ and then perform an assignment (2 instructions). Finally, at Line 10, we return a value (1 instruction). We have summarized this best-effort estimation in Figure 3.8.

Line in Contains	Number of instructions
Line 1	2
Line 2	2
Line 3	3
Line 4	1
Line 5	2
Line 7	2
Line 10	1

Figure 3.8: Counting the operations performed by the Contains algorithm.

Lines 2, Line 3, Line 4, Line 5, and Line 7 are part of the body of the while loop at Lines 2–9. Hence, each of these lines is repeated at-most *once* per value in list L (except for Line 2, which in addition is done once at the end of the loop). As such, execution of Contains takes at-most $5 + 10 \cdot N$ instructions. We counted both the if case of Lines 4–5 and the else case of Line 7, even though only one of these cases is executed per value in L. A closer look reveals that we execute Lines 4–5 only for those values $w \in L$ that are equivalent to v. Hence, if there are m copies of the value v in L, then execution of Contains takes exactly

$$\mathsf{NumInstr}(N) = 5 + 8 \cdot m + 7 \cdot (N - m) = 5 + 7 \cdot N + m \text{ instructions,}$$

assuming our estimate of the number of instructions for each line of the algorithm is correct.

The above model NumInstr is a bit hard to use: we need to know whether v is in L to use the model. For now, we assume that value v is not in list. Hence, we can simplify the above model to $\mathsf{NumInstrOE}(N) = 5 + 7 \cdot N + 0 = 5 + 7 \cdot N$ instructions.

Next, we will use the model NumInstrOE to make predictions. Assume that the runtime of CONTAINS on my laptop with a list L with $|L| = 1000$ values is 12 µs. Hence, NumInstrOE(1000) = 7005 instructions take 12 µs. If we *double* the size of L to $|L| = 2000$, then CONTAINS performs NumInstrOE(2000) = 14 005 instructions, which would take $\frac{14005}{7005} \cdot 12\,\mu s \approx 2 \cdot 12\,\mu s = 24\,\mu s$. Hence, we predict that *doubling* the input size will *double* the runtime. Likewise, we predict that increasing the size of L by a factor x will increase the runtime by a factor of x.

To verify whether our above predictions are correct, we can *implement* CONTAINS and compare our predictions to measurements on our implementation. Next, we detail a C++ implementation of CONTAINS together with a function to generate lists and a function to measure the performance of CONTAINS.

```
#include <chrono>
#include <cstddef>
#include <iostream>
#include <vector>

bool find(const std::vector<int>& list, int value)
{
    bool found = false;
    std::size_t i = 0;
    while (i != list.size()) {
        if (list[i] == value) {
            found = true;
            ++i;
        }
        else {
            ++i;
        }
    }
    return found;
}

std::vector<int> generate_list(std::size_t size)
{
    std::vector<int> table;
    while (table.size() != size) {
        table.emplace_back(table.size());
    }
    return table;
}

void measure_find(std::size_t size)
{
    using namespace std::chrono;
    auto list = generate_list(size);
    auto value = size; /* not in list. */

    auto start = steady_clock::now();
    find(list, value);
    auto end = steady_clock::now();
    auto measurement = duration_cast<microseconds>(end - start);

    /* Nicely print output. */
    std::cout << size << '\t' << measurement.count() << '\n';
}
```

```
int main(int argc, char* argv[])
{
    for (std::size_t size = 0; size <= 1024 * 1024 * 256; size += 4 * 1024 * 1024) {
        measure_find(size);
    }
}
```

We store this source code in a file `find.cpp` and we compiled the source code with the `g++` compiler (gcc version 13.1.0) using the command `g++ find.cpp -std=c++23 -O3` (here, `-std=c++23` specifies the version of C++ we are using and `-O3` instructs the compiler to fully optimize the code). In Figure 3.9, we visualized the measurements taken on a laptop.

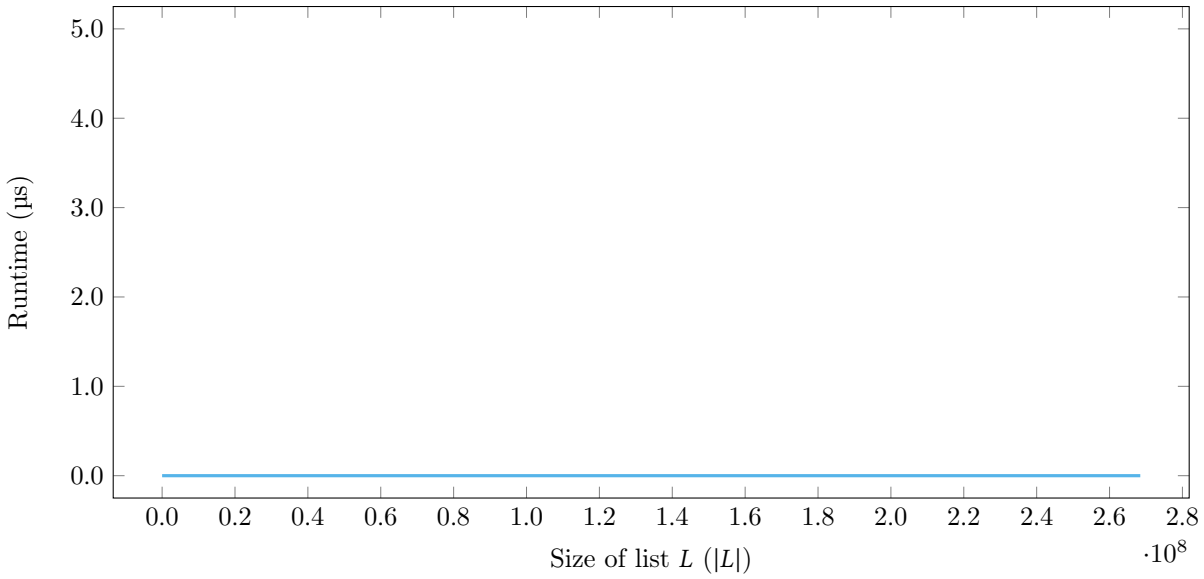


Figure 3.9: The measured performance of the CONTAINS algorithm implementation.

Based on Figure 3.9, our model and, hence, our predictions seem wrong. Indeed, according to the measurements, the runtime of CONTAINS is 0 µs *independent* of the size of the input. We are clearly doing something wrong!

The only two explanations for this are either that our CONTAINS implementation is *so fast* that we cannot measure it or that our CONTAINS implementation is *not executed at all*. To figure out which of the two it is, we instructed the compiler to generate a human-readable version of the executable instructions it produces (the *assembly output*). Below is the relevant part of this output:

```
call    _ZNSt6chrono3_V212steady_clock3nowEv
movq    %rax, %rsi
call    _ZNSt6chrono3_V212steady_clock3nowEv
```

This output indicates that function `std::chrono::steady_clock::now()` is executed twice (to obtain the values `start` and `end`). In between, we only move a value from one part of the processor to another (`movq` is a move instruction). Hence, we essentially do not do anything and the implementation *definitely does not* execute CONTAINS.

We stumbled upon a common pitfall when measuring the performance of an algorithm in isolation: we are *not* measuring the algorithm in the way it is used *in practice*. Hence, our measurement results might not line up with practical performance.

In this case, the implementation is simple enough due to which the compiler (which we instructed to *fully optimize* the code) can determine that some parts of the code effectively *do not do anything* (they do not

have side effects). This is the case for our usage of CONTAINS: we never use the output of CONTAINS and, hence, the compiler decides we do not need CONTAINS and removed it from our code.

Remark 3.14. Always make sure that performance measurements of your programs are performed in a way that reflects the *actual usage* of your program. Otherwise, your measurement results might not line up with the performance seen when using your program.

When comparing algorithms, special care needs to be taken to assure that the inputs of the algorithm *reflect* practical workloads. Furthermore, special care needs to be taken to assure that each measurement is performed in a fair and equal manner. For example, measuring the performance of *two* algorithms that both operate on a big list L (to figure out which one is faster) might give the *second* algorithm we measure an unfair advantage: the first algorithm already read values from L due to which these values might still be readily available to the processor (via caches) when measuring the performance of the second algorithm.

To deal with our measurement issue, we make a small change to the source code: we simply use the result of CONTAINS by printing out whether the call to CONTAINS found a value:

```
void measure_find(std::size_t size)
{
    using namespace std::chrono;
    auto list = generate_list(size);
    auto value = size; /* not in list. */

    auto start = steady_clock::now();
    auto found = find(list, value);
    auto end = steady_clock::now();
    auto measurement = duration_cast<microseconds>(end - start);

    /* Nicely print output. */
    std::cout << size << '\t' << measurement.count() << '\t' << found << '\n';
}
```

In Figure 3.10, we visualized the measurements taken on a laptop using this updated version of the function `measure_find`.

The measurements of Figure 3.10 do show some irregularities, e.g., the peak labeled A. These peaks are likely caused by the operating system deciding to run a quick background task (e.g., checking for updates or running a virus scanner). Practically, we could and should improve our measurements by repeating them multiple times, removing any outlying measurements, and then presenting the average. Such an improved statistical analysis of the measurements is outside the scope of these notes: the current measurements are good enough to validate whether our model made accurate predictions. In Figure 3.11, we have detailed the four data points B, C, D, and E.

As is clear from Figure 3.11, the *predictive value* of the model NumInstrOE is great: our measurements essentially match the predictions exactly.

With the above experiment, we have proven the worth of the NumInstrOE model. Hence, one might be inclined to think that the NumInstrOE model is a *good* model. This is not the case, however: a much simpler model exists that makes *exactly the same predictions*, namely the model

$$\mathsf{ContainsRuntime}(N) = N.$$

Even though the model ContainsRuntime is a very simple function-of-N, ContainsRuntime still models exactly the same runtime behavior as the models NumInstr or NumInstrOE.

Remark 3.15. Besides the existence of ContainsRuntime, a *simpler* model that makes exactly the same predictions as NumInstrOE, the model NumInstrOE is also fundamentally flawed in itself. With NumInstrOE, we set out to model the exact number of instructions performed by CONTAINS and use this number as a predictor of the runtime. This approach is wrong in two ways:

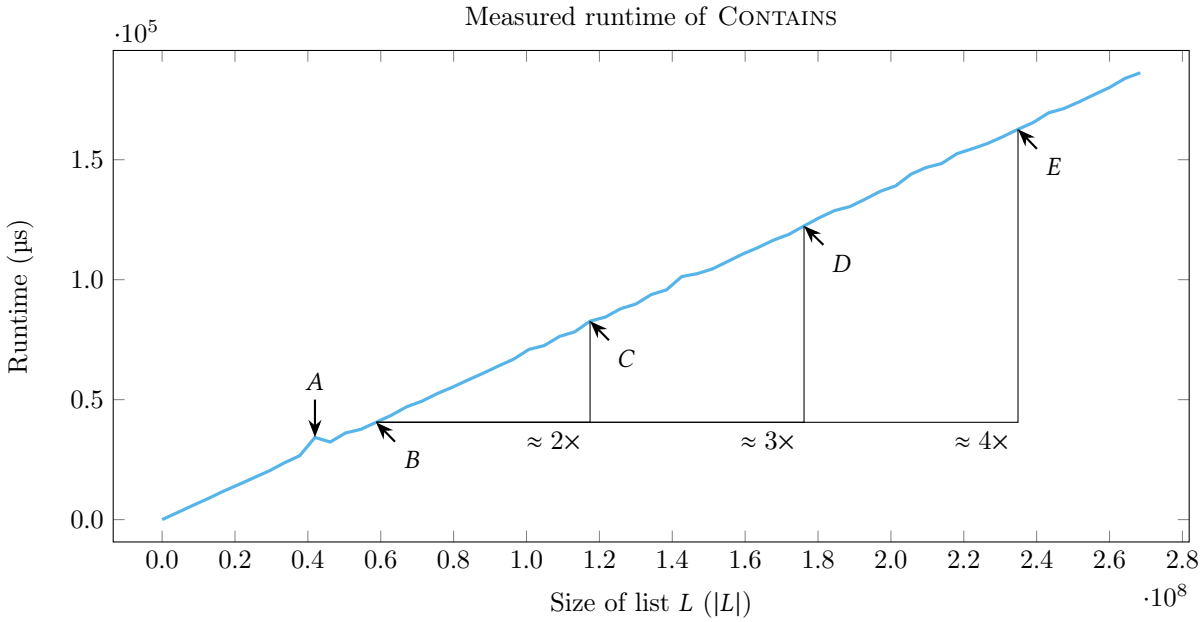


Figure 3.10: The measured performance of the updated Contains algorithm implementation.

Data point	Size of list L ($\lvert L\rvert$)		Runtime (µs)	
B	58 720 256		40 559 µs	
C	117 440 512	(2×)	82 730 µs	(≈ 2.04×)
D	176 160 768	(3×)	122 391 µs	(≈ 3.02×)
E	234 881 024	(4×)	162 682 µs	(≈ 4.01×)

Figure 3.11: Details on four data points taken from Figure 3.10.

1. First, the exact number of instructions used to execute Contains depends on many system-specific details. For example, our estimate of *two* instructions for Line 5 (the statement "$i := i + 1$" is wrong on most types of processors, as most processors have a dedicated *increment-by-one* instruction.

2. Second, we cannot simply translate an exact count of the number of instructions into a running time. In practice, not all instructions are equally expensive: some instructions can take orders-of-magnitudes longer than others to execute.

   For example, it is very likely that an implementation of Contains will keep the variables i, r, and v in processor registers (small pieces of extremely fast memory in the processor itself) and any instructions that *only* operate on these variables will be exceptionally fast. As a consequence, the one-instruction difference between either executing Lines 4–5 or executing Line 7 is irrelevant.

   At the same time, the instruction $L[i]$ on Line 3 of Contains reads the i-th value from array L. As array L can have many millions or even billions of values, one cannot store this array directly in the processor. Hence, the instruction $L[i]$ needs to read a value from *main memory*. On the laptop we used to measure the performance of Contains, reading a 4 byte integer from *main memory* takes roughly 11 ns, whereas the fastest instructions (e.g., those on Lines 4, 5, and 7) finish in less than 0.3 ns. Hence, our model should mainly be concerned with how often we execute the instruction $L[i]$.

Furthermore, the models NumInstr and NumInstrOE take into account the cost of Lines 1 and 10. These lines are only executed *once* and perform a handful of instructions. When Contains operates on large lists, the cost of these lines is irrelevant and only overcomplicates the model.

Taking Remark 3.15 into account, it is very reasonable to *remove* all unnecessary details from models

NumInstr and NumInstrOE, after which we end up with our preferred model ContainsRuntime. We conclude:

Proposition 3.16. *The runtime complexity of the* Contains *algorithm of Figure 3.1 on lists L of size $N = |L|$ is modelled by* $\mathsf{ContainsRuntime}(N) = N$.

Up till now, we have only considered the runtime complexity of Contains. We can also model the other costs of algorithms, e.g., the *memory usage* by the algorithm as a function of the size of the input. The Contains algorithm only uses a constant amount of memory besides the memory holding the input and output, namely memory to hold the variable i. We conclude:

Proposition 3.17. *The memory complexity of the* Contains *algorithm of Figure 3.1 on lists L of size $N = |L|$ is modelled by* $\mathsf{ContainsMemory}(N) = 1$.

Exercise 3.4. Make a model of the number of instructions executed by the ArraySum algorithm of Figure 1.1. Next, make a model of the runtime and memory usage of ArraySum.

3.1.4 Formalization of complexity

In the previous section, we constructed a model for the runtime of the Contains algorithm: we determined that the runtime of Contains is modelled by the model $\mathsf{ContainsRuntime}(N) = N$ with $N = |L|$ the size of the list L used as input of Contains. The ContainsRuntime model allowed us to make accurate predictions about the runtime of Contains, e.g., doubling the size of the input of Contains doubles the runtime. We have also seen that *other models* exist for Contains that model the same behavior (they allow us to make exactly the same predictions about the runtime of Contains), e.g., the models NumInstr and NumInstrOE.

Now consider we have an another algorithm AltC that also solves the *Contains problem* of Problem 3.1 and that the runtime of AltC is modelled by some model AltCRuntime. It is only natural to compare the runtime of Contains and AltC, e.g., to determine which algorithm is more efficient.

As we argued at the start of Section 3.1.3, we are mainly interested in *scalability* of algorithms: on small inputs, many algorithms are practically fast enough, but it takes effort to assure algorithms remain fast on huge inputs. We can use the models ContainsRuntime and AltCRuntime to predict how Contains and AltC will behave with larger inputs.

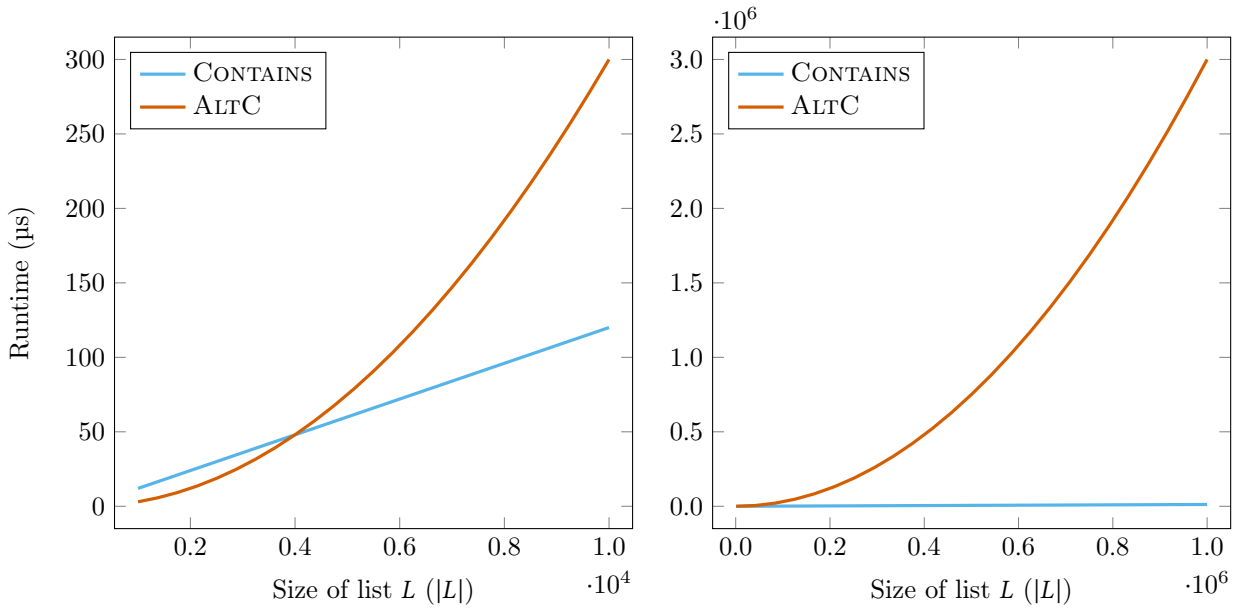
Example 3.18. We want to compare the algorithms Contains and AltC. We have $\textsc{ContainsRuntime}(N) = N$ and we assume that the runtime of Contains on a list L with $N = |L| = 1000$ values is 12 µs. For this example, consider $\mathsf{AltCRuntime}(N) = N^2$ and the runtime of AltC on a list L with $N = |L| = 1000$ values is 3 µs. Hence, AltC is significantly faster than Contains by a factor of $\frac{12\,\mu s}{3\,\mu s} = 4\times$.

Using the above information, we can predict the runtime of both algorithms when we *double* the input to a list with $|L| = 2000$ values. As determined in Section 3.1.3, we predict a runtime of 24 µs for Contains. To predict the runtime of AltC, we note that $\mathsf{AltCRuntime}(2000) = 2000^2 = 2^2 \cdot 1000^2 = 4 \cdot \mathsf{AltCRuntime}(1000)$. Hence, for AltC, we predict a runtime of 12 µs. As one can see, AltC is still faster than Contains, but now only by a factor of $\frac{24\,\mu s}{12\,\mu s} = 2\times$. In Figure 3.12, we have visualized the predicted runtimes for both Contains and AltC.

Even though AltC is faster than Contains on small lists, we predict that Contains will quickly outperform AltC when we increase the size of the input. This is easily explained by the models ContainsRuntime and AltCRuntime. The AltCRuntime model predicts a *faster growth* of the runtime than the ContainsRuntime model: e.g., doubling the input size only doubles the runtime of Contains, while it quadruples the runtime of AltC.

Consider two models $f(N)$ and $g(N)$ representing the runtime of algorithms AlgoF and AlgoG as a function of the size of the input N. In a similar way as illustrated in Example 3.18, we can compare AlgoF and AlgoG to determine which algorithm is *more efficient* (faster, has a lower runtime) on large inputs. As seen in Example 3.18, the conclusion of such a comparison is determined by the *order of growth* of $f(N)$ and $g(N)$: the algorithm whose model predicts a *slower growth* of the runtime will be the faster algorithm on large inputs.

Definition 3.19. Consider the functions f and g of n. We *informally* say the following:



Input size	1000	2000	*(2×)*	4000	*(4×)*	8000	*(8×)*	1 000 000	*(1000×)*
Runtime Contains	12 µs	24 µs	*(2×)*	48 µs	*(4×)*	96 µs	*(8×)*	12 ms	*(1000×)*
Runtime AltC	3 µs	12 µs	*(4×)*	48 µs	*(16×)*	192 µs	*(64×)*	3000 ms	*(1 000 000×)*
Speed up	4×	2×		1×		0.5×		0.004×	

Figure 3.12: A comparison of the predicted runtime of algorithms Contains and AltC as a function of the size of the input using the *known* runtimes for lists with 1000 values and using the models ContainsRuntime(N) = N and AltCRuntime(N) = N^2. *Top*, a visualization for a range of input sizes and *bottom*, the details for a few input sizes.

1. The order of growth of function f is *upper bounded* by g, denoted by $f(n) = \mathcal{O}(g(n))$, if f "scales better" than $g(n)$: any increase in the runtime predicted by f as a consequence of increasing the input size n is *at-most* the increase predicted by $g(n)$.

   For example, due to Example 3.18, we know that the order of growth for the model ContainsRuntime(N) = N is *upper bounded* by the model AltCRuntime(N) = N^2.

2. The order of growth of function f is *lower bounded* by g, denoted by $f(n) = \Omega(g(n))$, if f "scales worse" than $g(n)$: any increase in the runtime predicted by f as a consequence of increasing the input size n is *at-least* the increase predicted by $g(n)$.

   For example, due to Example 3.18, we know that the order of growth for the model AltCRuntime(N) = N^2 is *lower bounded* by the model ContainsRuntime(N) = N.

3. The order of growth of function f is *equivalent* to g, denoted by $f(n) = \Theta(g(n))$, if f "scales the same" as $g(n)$: any increase in the runtime predicted by f as a consequence of increasing the input size n is *equivalent to* the increase predicted by $g(n)$. In this case, we also say that $f(n)$ is *strictly bounded by* $g(n)$.

   For example, as argued in Section 3.18, we know that the order of growth for the model NumInstrOE(N) = $3 + 7 \cdot N$ is *equivalent* to the order of growth of models ContainsRuntime(N) = N.

With the terminology of Definition 3.21, we can rephrase Proposition 3.17:

Theorem 3.20. *The runtime complexity of the* Contains *algorithm on lists of length N is $\Theta(N)$. The memory complexity of the* Contains *algorithm on lists of length N is $\Theta(1)$.*

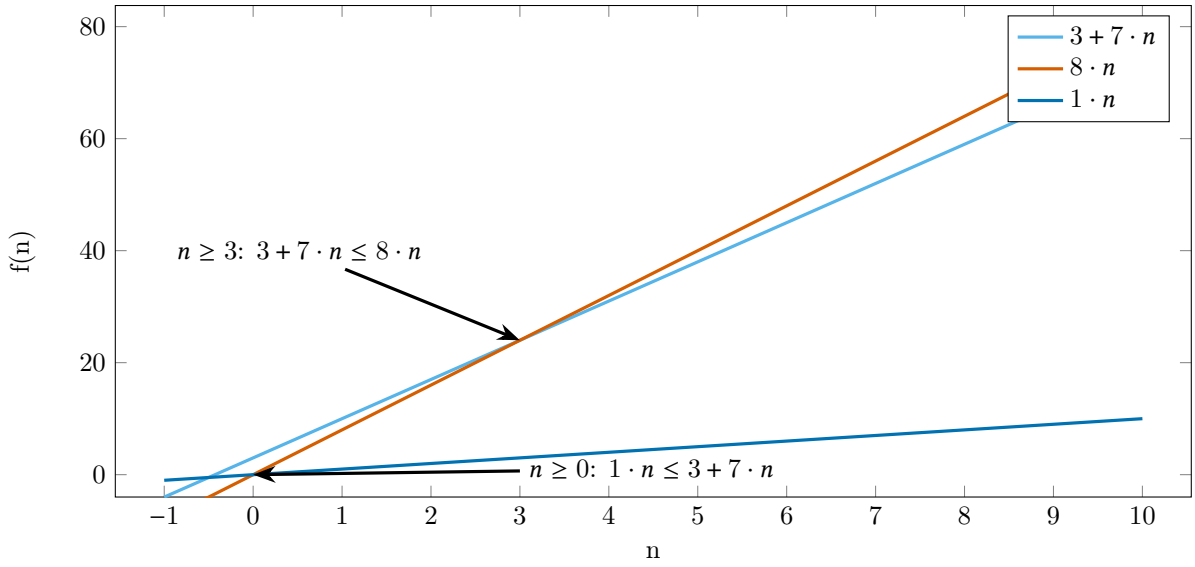


Figure 3.13: Visualization of the proof for $3 + 7 \cdot n = \Theta(n)$. We refer to Example 3.22 for details.

Definition 3.19 is rather informal, making it hard to apply the definition in practice. Next, we formalize the notions introduced in Definition 3.19:

Definition 3.21. Consider the functions f and g of n. We have

1. $f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that, for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$;
2. $f(n) = \Omega(g(n))$ if there exists constants $n_0, c > 0$ such that, for all $n \geq n_0$, $0 \leq c \cdot g(n) \leq f(n)$;
3. $f(n) = \Theta(g(n))$ if there exists constants $n_0, c_{\text{lb}}, c_{\text{ub}} > 0$ such that, for all $n \geq n_0$, $0 \leq c_{\text{lb}} \cdot g(n) \leq f(n) \leq c_{\text{ub}} \cdot g(n)$.

Example 3.22. Next, we shall show that $3+7 \cdot n = \Theta(n)$ by showing $3+7 \cdot n = \mathcal{O}(n)$ and by showing $3+7 \cdot n = \Omega(n)$.

- $3 + 7 \cdot n = \mathcal{O}(n)$. Choose $n_0 = 3$ and $c = 8$. The statement

  $$\text{for all } n \geq 3,\ 0 \leq 3 + 7 \cdot n \leq 8 \cdot n$$

  is true, completing the proof.

- $3 + 7 \cdot n = \Omega(n)$. Choose $n_0 = 0$ and $c = 1$. The statement

  $$\text{for all } n \geq 0,\ 0 \leq 1 \cdot n \leq 3 + 7 \cdot n$$

  is true, completing the proof.

We have visualized the details of this proof in Figure 3.13.

Remark 3.23. It is always a good idea to reflect on formal definitions to make sure we understand the definition and we understand what each part of the definition represents. We do so for the definition of $f(N) = \mathcal{O}(g(N))$. According to Definition 3.21, we have

> "$f(n) = \mathcal{O}(g(n))$ if there exists constants $n_0, c > 0$ such that, for all $n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$".

This definition introduces constants n_0 and c. What is the *conceptual* meaning of these constants?

Assume f and g are models for the runtime of algorithms AlgoF and AlgoG respectively. In this setting, we use n to represent the size of the input. Hence, the constant n_0 restricts the input sizes we consider in our comparison of the functions f and g: we are looking at the behavior of f and g with respect to large

inputs, as we do not look at small inputs (those smaller than n_0). To understand why the definition does this, it is important to remember that we are mainly interested in the scalability of AlgoF and AlgoG. As illustrated in Example 3.18 and Figure 3.12, models with a slow growth predict *better scalability* than models with a high growth: ContainsRuntime predicted better scalability than AltCRuntime. Models with a slow growth do not necessary predict *lower runtimes*, however: they only do so if we make the input sufficiently large.

The constant n_0 also simplifies the definition of a model: some functions behave weird for small inputs, e.g., $\frac{1}{n}$ and $\log_2(n)$ are not defined for $n := 0$. We do not have to consider such inputs by choosing $n_0 > 0$.

The role of constant c has already been illustrated in Section 3.1.3, where we saw that the models $\mathsf{ContainsRuntime}(N) = N$ and $\mathsf{NumInstrOE}(N) = 3 + 7 \cdot N$ made exactly the same predictions: the constant c allows us to compare models that are different due to non-essential factors (e.g., a different way in which the number of instructions are accounted for).

From straightforward application of Definition 3.21, we can derive the following technical results:

Lemma 3.24. *We have $f(n) = \mathcal{O}(g(n))$ if and only if $g(n) = \Omega(f(n))$ and we have $f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.*

Determining how the order of growth of $f(n)$ relates to the order of growth of $g(n)$ is, fundamentally, a question that can be answered by *mathematics*. For example:

Theorem 3.25. *Let f and g be functions of n with non-negative ranges. If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \text{ is defined and is } \begin{cases} \infty & \text{then } f(n) = \Omega(g(n)); \\ c, \text{ with } c > 0 \text{ a constant} & \text{then } f(n) = \Theta(g(n)); \\ 0 & \text{then } f(n) = \mathcal{O}(g(n)). \end{cases}$$

Using Theorem 3.25, we derive key results to help us compare the order of growth of common functions.

Example 3.26. We have $\lim_{n\to\infty} \frac{5\cdot n}{n} = 5$. Hence, $5 \cdot n = \Theta(n)$ More generally, for every constant $c > 0$, we have

$$\lim_{n\to\infty} \frac{c \cdot f(n)}{f(n)} = c \cdot \left(\lim_{n\to\infty} \frac{f(n)}{f(n)}\right) = c. \text{ Hence, } \boxed{c \cdot f(n) = \Theta(f(n))}.$$

As $\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = \frac{1}{\log_b(a)} \cdot \log_b(n)$ and $\frac{1}{\log_b(a)}$ is a constant, we have $\boxed{\log_a(n) = \Theta(\log_b(n))}$.

We have $\lim_{n\to\infty} \frac{n^2}{n} = \lim_{n\to\infty} n = \infty$ and we have $\lim_{n\to\infty} \frac{n}{n^2} = 0$. Hence, $n^2 = \Omega(n)$ and $n = \mathcal{O}(n^2)$. More generally, for every constant $c, d > 0$ we have

$$\lim_{n\to\infty} \frac{n^c}{n^{c+d}} = \lim_{n\to\infty} \frac{1}{n^d} = 0. \text{ Hence, } \boxed{n^c = \mathcal{O}(n^{c+d})}.$$

We have $\lim_{n\to\infty} \frac{\log_2(n)^{100}}{\sqrt{n}} = 0$. Hence, $\log_2(n)^{100} = \mathcal{O}(\sqrt{n})$. More generally, for any constants $c > 0, d > 0$, we have

$$\lim_{n\to\infty} \frac{\log_2(n)^c}{n^d} = 0. \text{ Hence, } \boxed{\log_2(n)^c = \mathcal{O}(n^d)}.$$

We have $\lim_{n\to\infty} \frac{n^{100}}{1.1^n} = 0$. Hence, $n^{100} = \mathcal{O}(1.1^n)$. More generally, for any constants $c > 0, d > 1$, we have

$$\lim_{n\to\infty} \frac{n^c}{d^n} = 0. \text{ Hence, } \boxed{n^c = \mathcal{O}(d^n)}.$$

We have $\lim_{n\to\infty} \frac{2^n}{3^n} = 0$ and $\lim_{n\to\infty} \frac{2^{n/2}}{2^n} = 0$. Hence, $2^n = \mathcal{O}(3^n)$ and $2^{n/2} = \mathcal{O}(2^n)$. More generally, for any constants $c \geq d \geq 1, u \geq v \geq 1$, we have

$$\lim_{n\to\infty} \frac{d^{n/u}}{c^{n/v}} = 0. \text{ Hence, } \boxed{d^{n/u} = \mathcal{O}(c^{n/v})}.$$

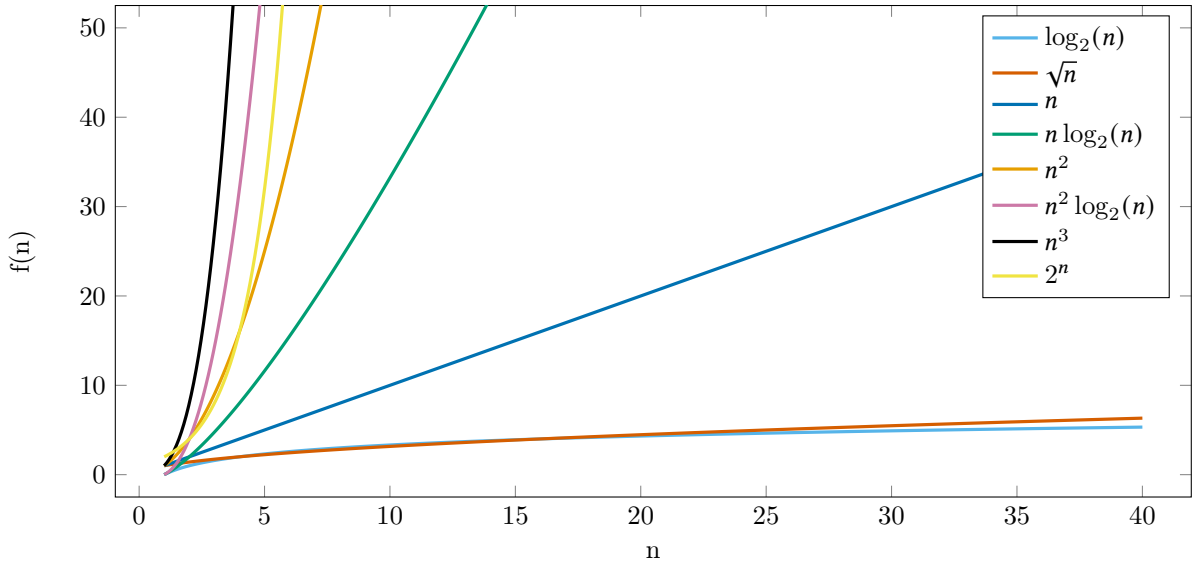


Figure 3.14: Typical complexity functions of n that we could encounter in these notes. Although not clear from this figure, 2^n will eventually overtake n^3 (namely for $n = 10$, we have $2^{10} = 1024$ and $n^3 = 1000$).

Finally, we have $\lim_{n\to\infty} \frac{3\cdot n^2+7\cdot n}{n^2} = 3 \cdot (\lim_{n\to\infty} \frac{n^2}{n^2}) + 7(\lim_{n\to\infty} \frac{n}{n^2}) = 3 + 0 = 3$. Hence, $3 \cdot n^2 + 7 \cdot n = \Theta(n^2)$. More generally, consider a polynomial of the form $c_1 \cdot n^{d_1} + \cdots + c_m \cdot n^{d_m}$ with constants $c_1, \ldots, c_m, d_1, \ldots, d_m > 0$. We have

$$\lim_{n\to\infty} \frac{c_1 \cdot n^{d_1} + \cdots + c_m \cdot n^{d_m}}{n^{d_i}} = c_i \text{ with } d_i = \max(d_1, \ldots, d_m). \text{ Hence, } \boxed{c_1 \cdot n^{d_1} + \cdots + c_m \cdot n^{d_m} = \Theta(n^{d_i})}.$$

Assume $f(n) = \mathcal{O}(g(n))$. We also have

$$\lim_{n\to\infty} \frac{f(n)+g(n)}{g(n)} = \left(\lim_{n\to\infty} \frac{f(n)}{g(n)}\right) + \left(\lim_{n\to\infty} \frac{g(n)}{g(n)}\right) = 0 + 1 = 1. \text{ Hence, } \boxed{f(n) + g(n) = \Theta(g(n))}.$$

Similarly, if $f(n) = \mathcal{O}(g(n))$, then we have

$$\lim_{n\to\infty} \frac{h(n)\cdot f(n)}{h(n)\cdot g(n)} = \left(\lim_{n\to\infty} \frac{h(n)}{h(n)}\right) \cdot \left(\lim_{n\to\infty} \frac{f(n)}{g(n)}\right) = \left(\lim_{n\to\infty} \frac{f(n)}{g(n)}\right) = 0. \text{ Hence, } \boxed{h(n)\cdot f(n) = \mathcal{O}(h(n)\cdot g(n))}.$$

For example, $n^2 \log_2(n) = \mathcal{O}(n^3)$ because $\log_2(n) = \mathcal{O}(n)$.

In these notes, we will often apply the rules of thumb highlighted in Example 3.26 without further explanation. We do not require further knowledge of limits in these notes. Sufficient familiarity with limits can help with deriving rules of thumb for such as the one introduced in Example 3.26 for functions we did not highlight in the example, however.

Typical functions used to model runtime complexity use logarithms, polynomials, exponential functions, and combinations of these. The rules of thumbs of Example 3.26 are sufficient to determine the relationship between the order of growth of such functions. In Figure 3.14, we have sketched the typical complexity functions that we will frequently encounter in these notes.

Exercise 3.5. What is the runtime complexity of the ArraySum algorithm of Figure 1.1?

Exercise 3.6. Order the following functions of n on increasing growth rates and group functions with identical growth rates. Explain your answers.

$\ln(n^3)$	n	$n^2\log_2(n)$	$4^{\log_2(n)}$	$\log_2(\sqrt{n})$	$2n\log_2(n)$
$n + \log_2(n^4)$	$2^{\log_2(16)}$	n^{-1}	16	$n^{\log_2(4)}$	$\log_2(n^n)$

3.2 The Linear Search Algorithm

In Section 3.1, we introduced the CONTAINS algorithm, proved it to be correct, and showed that it had a runtime complexity of $\Theta(|L|)$. Given these facts, we might be inclined to believe that CONTAINS is a *good* algorithm. From an algorithm design perspective, this is not entirely true: the *contains problem* (Problem 3.1) is *too* specialized due to which we cannot use CONTAINS for anything besides solving the contains problem.

As a more-flexible alternative to the *contains problem*, we next consider the *search problem*:

Problem 3.27. Let L be a list, v be a value, and o, $0 \leq o < |L|$, be an offset in list L. A solution to the *search problem* for L, v, and o will compute the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

The search problem can be solved with the LINEARSEARCH algorithm of Figure 3.15.

Algorithm LINEARSEARCH(L, v, o) **:**
Pre: L is an *array*, v a value, $0 \leq o \leq |L|$.
1: $r := o$.
   /* invariant: "$o \leq r \leq |L|$ and $v \notin L[o, r)$", bound function: $|L| - r$ */
2: **while** $r \neq |L|$ **and also** $L[r] \neq v$ **do**
3: $\quad r := r + 1$.
4: **end while**
5: **return** r.
Post: return the first offset r, $o \leq r < |L|$, with $L[r] = v$ or, if no such offset exists, $r = |L|$.

Figure 3.15: The LINEARSEARCH algorithm.

The LINEARSEARCH algorithm is not too different from a simplified version of the CONTAINS algorithm of Figure 3.1: the biggest change being that we can now start our inspection of list L at any offset o (instead of always starting at the first value $L[0]$). We leave it as an exercise to the reader to prove the following:

Theorem 3.28. *The LINEARSEARCH algorithm of Figure 3.15 is correct and solves Problem 3.27. The complexity of the LINEARSEARCH algorithm is* $\mathcal{O}(|L|)$.

The LINEARSEARCH algorithm is flexible: we can easily use it as a building block in other algorithms. For example, we can solve the *contains problem* with the single-line algorithm LSCONTAINS of Figure 3.16, *top*, that only returns the value LINEARSEARCH$(L, v, 0) \neq |L|$. Likewise, we can use the LINEARSEARCH algorithm to *count* the number of occurrences of a value v in a list L, e.g., the COUNT algorithm that can be found in Figure 3.16, *bottom*.

We now have *two algorithms* for solving the contains problem of Problem 3.1: the CONTAINS algorithm studied in Section 3.1 and the LSCONTAINS algorithm of Figure 3.16. We already know that the runtime complexity of CONTAINS is $\Theta(|L|)$. Next, we will look at the complexity of LSCONTAINS. Obviously, the runtime complexity of LSCONTAINS is determined entirely by the runtime complexity of LINEARSEARCH.

According to Theorem 3.28, the runtime complexity of LINEARSEARCH is $\mathcal{O}(|L|)$. This upper bound is a simplification of reality, however: for many inputs of LINEARSEARCH, the runtime complexity will be low. Hence, it makes sense to refine our analysis of LINEARSEARCH. In a first attempt to do so, we will look at the best case, average case, and worst case complexity of LINEARSEARCH.

Definition 3.29. Let ALGO be an algorithm.

1. We say that the *best case runtime complexity* of ALGO, as a function of the input size n, is modelled by $\Theta(f(n))$ if, for every n, there are inputs of ALGO such that the runtime complexity of ALGO with respect to only these inputs is $\Theta(f(n))$.

2. We say that the *worst case runtime complexity* of ALGO, as a function of the input size n, is $\Theta(f(n))$ if, for every n, there are inputs of ALGO such that the runtime complexity of ALGO with respect to only these inputs is $\Theta(f(n))$.

Algorithm LSContains(L, v) **:**
Pre: L is an *array*, v a value.
1: **return** LinearSearch$(L, v, 0) \neq |L|$.
Post: return `true` if $v \in L$ and `false` otherwise.

Algorithm Count(L, v) **:**
Pre: L is an *array*, v a value.
2: $c, i := 0, 0$.
/* invariant: c is the count of v in $L[0, i)$, bound function: $|L| - i$ */
3: **while** `true` **do**
4: $i :=$ LinearSearch(L, v, i).
5: **if** $i = |K|$ **then**
6: **return** c.
7: **else**
8: $c, i := c + 1, i + 1$.
9: **end if**
10: **end while**
Post: return the number of copies of v in L.

Figure 3.16: The LSContains and Count algorithms that are both written in terms of the LinearSearch algorithm of Figure 3.15.

3. We say that the *average case runtime complexity* of Algo, as a function of the input size n, is modelled by $\Theta(f(n))$ if the runtime complexity of Algo is $\Theta(f(n))$ when averaged out over all possible inputs of size n of Algo.

Besides talking about strict bounds $\Theta(f(n))$, we can also talk about best case, worst case, and average case lower bounds and about best case, worst case, and average case upper bounds.

For every size $|L|$ of list L, the best case of LinearSearch is searching for a value v that is *at the offset* o of L (searching for the value $v := L[o]$). In this case, the LinearSearch algorithm will always perform only a handful of instructions. Hence, the best case runtime complexity of LinearSearch is $\Theta(1)$.

For every size $|L|$ of list L, the worst case of LinearSearch is searching for a value v that is *not in* L starting at offset $o = 0$. In this case, the LinearSearch algorithm will inspect every possible value in L. Hence, the worst case runtime complexity of LinearSearch is $\Theta(|L|)$.

Remark 3.30. For many algorithms, the best case, worst case, and average case complexity are *identical*, e.g., the Contains algorithm of Figure 3.1. To emphasize that these three cases have the same complexity, we sometimes refer to the *all case* complexity in these notes.

For other algorithms, finding the best case or worst case complexity might be easy, while finding the average case complexity is often hard. For example, there is no clear way to average out the unbounded number of inputs for which LinearSearch shows the best case behavior with the unbounded number of inputs for which LinearSearch shows the worst case behavior.

The best case and worst case runtime complexity of LinearSearch are rather different. This is indicative of a limitation of our runtime model: we are modeling the runtime of LinearSearch only in terms of the size of list L. As we have seen, the size of list L not always has influence on the runtime of LinearSearch. Hence, we can do better by modelling the runtime of LinearSearch in terms of other variables: if we are looking for the i-th value in list $L[o, |L|)$, then the *all case* complexity of LinearSearch will be $\Theta(i)$. Unfortunately, we *cannot* express i solely in terms of L (or in terms of the other inputs v and o). We can express i in terms of the input o and output $r :=$ LinearSearch(L, v, o), however: an exact model for the complexity of LinearSearch is $\Theta(r - o)$.

Remark 3.31. Exact models of the runtime of an algorithm are often hard to obtain. Hence, it is much more common to provide a *worst case upper bound* on the runtime of algorithms. Furthermore, the runtime

complexity is *not always* modelled in terms of the input: in these notes, we will see several practical data processing algorithms whose complexity is *mainly* determined by the output size (and *not* by the input size).

Based on our analysis of LinearSearch, we can conclude that LSContains will perform less work than Contains *whenever* $v \in L$. Otherwise, when $v \notin L$, the two algorithms behave similar.

Next, we look at the Count algorithm. If we assume that LinearSearch is correct, then that will greatly simplify the proof of correctness for Count. This is a good thing: the correctness of LinearSearch is independent of the Count. Hence, we can prove the correctness of LinearSearch without considering Count: we only need to prove the correctness of LinearSearch *once* and we can reuse that correctness result in all algorithms that utilize LinearSearch. This reuse is an important consequence of proper *modular software design*.

Remark 3.32. When designing software, it is a good idea to strive for *modularity*: one should always strive to break up complex problems into a set of smaller subproblems. Typically, these subproblems are easier to analyze, easier to solve on themselves, and easier to debug on themselves. After ensuring each subproblem is solved very well by dedicated functions, solving the original complex problem can be done by simply combining these subproblem-solving functions. Furthermore, an independent function that solves one of these smaller subproblems might, on its own, provide useful functionality that you can reuse elsewhere in your code.

Now lets look at the complexity of Count. The variable i is increased by at-least *one* at each iteration of the loop. Hence, Count will call LinearSearch (Line 4) at-most $|L|$ times. By Theorem 3.28, the cost of the call to LinearSearch is $\mathcal{O}(|L|)$. Hence, the complexity of Count is upper-bounded by $\mathcal{O}(|L|^2)$. With a bit more effort, we can provide a much better analysis of the runtime complexity of Count, however. In Count, consecutive calls to LinearSearch (Line 4) will inspect consecutive parts of the list L. Hence, the complexity of Count is $\Theta(|L|)$.

⋆ *Remark* 3.33. The LinearSearch algorithm can be further generalized: what if we want to search for the first value that is smaller-than-v, larger-than-v, a multiple-of-v, or an odd number (not divisible by two). The pseudo-code for such search algorithms is almost identical to the LinearSearch algorithm in Figure 3.15: only the condition "$L[r] \neq v$" in the while loop at Line 2 needs to change. Many programming languages allow one to write a general recipe for all these search algorithms by providing the ability to pass as argument a *search predicate function* that returns `true` for the values one is searching for. In Figure 3.17, we have illustrated this approach as an algorithm.

Algorithm LinearPredSearch(L, P, o) **:**

Pre: L is an *array*, $0 \leq o \leq |L|$, and P is a predicate on values.

1: $r := o$.
   /* invariant: "$o \leq r \leq |L|$ and $\neg(P(v))$ for all $v \in L[o, r)$", bound function: $|L| - r$ */
2: **while** $r \neq |L|$ **and also** $\neg(P(L[r]))$ **do**
3: $\quad r := r + 1$.
4: **end while**
5: **return** r.

Post: return the first offset r, $o \leq r < |L|$, with $P(L[r])$ or, if no such offset exists, $r = |L|$.

Algorithm LinearSearch(L, v, o) **:**

6: $P(x)$ returns `true` if and only if $x = v$.
7: **return** LinearPredSearch(L, P, o).

Algorithm SearchFirstMultipleOf(L, v, o) **:**

8: $P(x)$ returns `true` if and only if x is a multiple of v.
9: **return** LinearPredSearch(L, P, o).

Figure 3.17: The LinearPredicateSearch algorithm and its usage.

Exercise 3.7. Assume that the LinearSearch algorithm of Figure 3.15 is correct. Prove that the LSContains algorithm of Figure 3.16 is correct.

Algorithm LowerBoundRec(L, v, *begin*, *end*) **:**
Pre: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.

```
1: if begin = end then
2:   return begin.
3: else
4:   mid := (begin + end) div 2.
5:   if L[mid] < v then
6:     return LowerBoundRec(L, v, mid + 1, end).
7:   else {L[mid] ≥ v}
8:     return LowerBoundRec(L, v, begin, mid).
9:   end if
10: end if
```

Post: return the first offset r, $begin \leq r < end$, with $v \leq L[r]$ or, if no such offset exists, $r = end$.

Figure 3.18: The LowerBoundRec algorithm that uses *binary searching* to find the first position p in L with $v \leq L[p]$.

Exercise 3.8. Prove that the LinearSearch algorithm of Figure 3.15 is correct.

Exercise 3.9. Assume that the LinearSearch algorithm of Figure 3.15 is correct. Prove that the Count algorithm of Figure 3.16 is correct.

Exercise 3.10. Assume that the LinearSearch algorithm is *only used* with inputs L and v such that $v \in L$. Hence, LinearSearch is only used to find the offset of values v that are in L. What is the *average case* complexity of LinearSearch under this assumption?

3.3 Binary Searching

The LinearSearch algorithm is a versatile algorithm for finding a value in any list. Unfortunately, it is also rather slow: if we often want to check whether a value is in a list L, then LinearSearch will do so at a potentially-high cost. We cannot easily do better, however: if we do not know anything about what is in a list L, then the only way to determine whether $v \in L$ is to inspect each element. Hence, if we often have to search in a list, then it would be helpful if we *know more about the list* before we start searching.

Fortunately, many datasets have implicit or explicit structures embedded in them. Take, for example, a list of all students enrolled for a course: if we maintain that list by always adding newly enrolled students to the end of the list, then the list of enrolled students is automatically ordered on *increasing enrollment date*.

Next, we will take a look at how to answer the search problem *assuming* that the input is ordered. Let L be an ordered list and assume we are looking for a value v. If we *compare* the value $L[i]$, $0 \leq i < |L|$, with value v, then we have three possible outcomes:

1. $L[i] < v$: as the list L is ordered, we know that every value in the list $L[0, i]$ is smaller than v. Hence, $v \in L$ if and only if $v \in L[i + 1, |L|)$.
2. $L[i] = v$: we have found the value v.
3. $L[i] > v$: as the list L is ordered, we know that every value in the list $L[i, |L|)$ is larger than than v. Hence, $v \in L$ if and only if $v \in L[0, i)$.

Hence, if L is ordered, then a *single comparison* can help us to exclude a large part of the list. To maximize the part of the list we can exclude following a comparison, we can always opt to inspect the *middle* of the list and then perform the next comparison on the remainder of the list. This process is called *binary searching*. We refer to Figure 3.18 for the LowerBoundRec algorithm, a variant of binary searching that always returns the *first position* p in list L with $v \leq L[p]$.

The LowerBoundRec algorithm is the first *recursive algorithm* in these notes. Hence, we have not yet seen how to analyze recursive algorithms (e.g., prove their correctness and determine their runtime and memory complexity).

Recursion is typically used to *repeat* an algorithm many times, this similar to the purpose of a while loop. Hence, as with correctness proofs involving while loops, the proof technique that seems most suitable for reasoning about recursion is *induction*. In specific, we will perform induction on the size of the part of the list we are inspecting, hence, on $(end - begin)$.

As we are using induction, we have to prove a base case, formalize an induction hypothesis, and prove the inductive step:

- *Base case*. The base case of a recursive algorithm are those cases in which execution of the algorithm *does not* perform recursive calls. Hence, the base case of LowerBoundRec is the case in which we are done inspecting all parts of the list ($end - begin = 0$). In this case, we only execute Lines 1–2 and we return *begin* which, in this case, is equivalent to *end*. As no offset r can exist with $begin \leq r < end$ and we return the value *end*, the post-condition holds.

- *Induction hypothesis*. For any list L', any value v', and any offsets $0 \leq begin' \leq end' \leq |L'|$ with $j = end' - begin'$, $0 \leq j < m$, the call LowerBoundRec(L', v', $begin'$, end') returns a result r that satisfies the post-condition.

- *Inductive step*. The inductive step of a recursive algorithm are those cases in which execution of the algorithm *performs* recursive calls. Hence, the inductive step of LowerBoundRec are calls LowerBoundRec(L, v, *begin*, *end*) with $0 < m = end - begin$. In this case, we are *not done* inspecting the list and we execute Lines 3–10. As $0 < end - begin$, we have $begin \leq mid < end$ after Line 4.

  We assume that the induction hypothesis holds: for recursive calls to LowerBoundRec that inspect less-than m values in list L, we expect the correct result. Based on the if-statement on Line 5, we distinguish two cases:

  1. *The condition $L[mid] < v$ holds*. As L is ordered, the result must be an offset r with $mid < r \leq end$. In this case, we perform the recursive call LowerBoundRec($L, v, mid + 1, end$) at Line 6. As we have $begin \leq mid < end$, we can conclude $(end - (mid + 1)) < m$. Hence, by the induction hypothesis, the call LowerBoundRec($L, v, mid + 1, end$) will return the correct offset r and the post-condition holds.
  2. *The condition $L[mid] < v$ does not hold*. Hence $v \leq L[mid]$. As L is ordered, the result must be an offset r with $begin \leq r \leq mid$. In this case, we perform the recursive call LowerBoundRec(L, v, $begin, mid$) at Line 8. As we have $begin \leq mid < end$, we can conclude $(mid - begin) < m$. Hence, by the induction hypothesis, the call LowerBoundRec($L, v, begin, mid$) will return the correct offset r and the post-condition holds.

As with while loops, we have to prove that the recursion eventually terminates by reaching a base case. The above proof already did so: each recursive calls will inspect a smaller portion of the list. Hence, eventually, we will reach an empty portion of the list (the base case, which does no further recursive calls).

Proposition 3.34. *The LowerBoundRec algorithm of Figure 3.18 is correct.*

Next, we shall work toward determining the runtime complexity of LowerBoundRec such that we can determine whether LowerBoundRec is an improvement over LinearSearch.

The complexity of most recursive algorithms is determined by four factors:

1. the amount of work the algorithm performs in the base cases;
2. the amount of work the algorithm performs in the recursive cases (excluding the work done by recursive calls);
3. the number of recursive calls performed during a single execution of a recursive case; and
4. the size of the input for these recursive calls.

One can typically summarize these factors using a recurrence of the form

$$T(n) = \begin{cases} f_1(n) & \text{if } n \leq M \text{ (\textit{base cases})}; \\ c \cdot T(g(n)) + f_2(n) & \text{if } n > M \text{ (recursive cases)}, \end{cases}$$

in which n is the size of the input, M is the maximum size of inputs handled by the base cases, $f_1(n)$ is a model for the amount of work performed in the base case, $f_2(n)$ is the amount of work performed in the recursive cases, c is the number of recursive calls performed during a single execution of a recursive case of the algorithm, and $g(n)$ is a function that provides the size of the input of these recursive calls.

Example 3.35. Consider a call LowerBoundRec(L, v, *begin*, *end*) of the LowerBoundRec algorithm. Let $n = (end - begin)$ be the size of the part of the list L inspected by this call.

In any call of LowerBoundRec, both the base case and the recursive case perform a constant amount of work (testing the condition of if-statements and, in the recursive case, computing *mid*). Hence $f_1(n) = 1$ and $f_2(n) = 1$. Furthermore, in the recursive case of LowerBoundRec, the algorithm performs one recursive call (either at Line 6 or at Line 8). Hence, $c = 1$. The size of these recursive calls is at-most-half the size of the original input. Hence, $g(n) = \lfloor \frac{n}{2} \rfloor$. We conclude that the runtime complexity of LowerBoundRec is modelled by the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 0; \\ 1 \cdot T\left(\lfloor \frac{n}{2} \rfloor\right) + 1 & \text{if } n \geq 1, \end{cases}$$

in which $n = (end - begin)$ is the size of the part of the list L inspected by LowerBoundRec.

In most cases, the amount of work performed in the base cases is upper bounded by some constant ($f_1(n) = \Theta(1)$). To simplify the notation, one often omits the base cases for such recurrences.

Example 3.36. The runtime complexity of LowerBoundRec is modelled by the recurrence

$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1.$$

A recurrence $T(n)$ can be viewed as a very high-level abstraction of a recursive algorithm that *only* contains the information relevant for determining the runtime complexity. The next step is to simplify the recurrence to an equivalent *closed form*: by finding a 'normal function' $h(n)$ of n (e.g., functions such as $h(n) = \log(n)$, $h(n) = n \log n$, or $h(n) = n^2$) that is *equivalent* to $T(n)$.

One typically uses induction to prove that $h(n)$ is the closed form of a recurrence $T(n)$. Induction does not help you find $h(n)$, however: you can only try to prove that $T(n) = h(n)$ using induction after you *guessed* $T(n) = h(n)$, as the function $h(n)$ will be a crucial part of the induction hypothesis.

Luckily, several methods exist to obtain the closed form of a recurrence without resorting to an inductive proof. Here, we will look at one such method: a recursion tree.

Definition 3.37. Let Algo be a recursive algorithm. The structure of the recursion tree $\mathcal{T}(n)$ for the runtime complexity of Algo with input size n is as follows:

1. Each node represents a call to Algo and is labeled with the input size of that call.
2. The root of the recursion tree $\mathcal{T}(n)$ represents the call to Algo with input size n.
3. If a node $u \in \mathcal{T}(n)$ has no children, then u represents a base case of Algo.
4. If a node $u \in \mathcal{T}(n)$ has children, then u represents a recursive case of Algo and the children of u represent all the recursive calls performed during the single execution of the recursive case represented by node u.

In the recursion tree $\mathcal{T}(n)$, each node $u \in \mathcal{T}(n)$ is annotated with the amount of work work(u) done by the call to Algo represented by that node (excluding the work done by recursive calls).

To obtain the total amount of work done by Algo, one simply computes the sum $\sum_{u \in \mathcal{T}(n)} \text{work}(u)$. If the recursion tree is sufficiently simple, one can compute the sum $\sum_{u \in \mathcal{T}(n)} \text{work}(u)$ by determining:

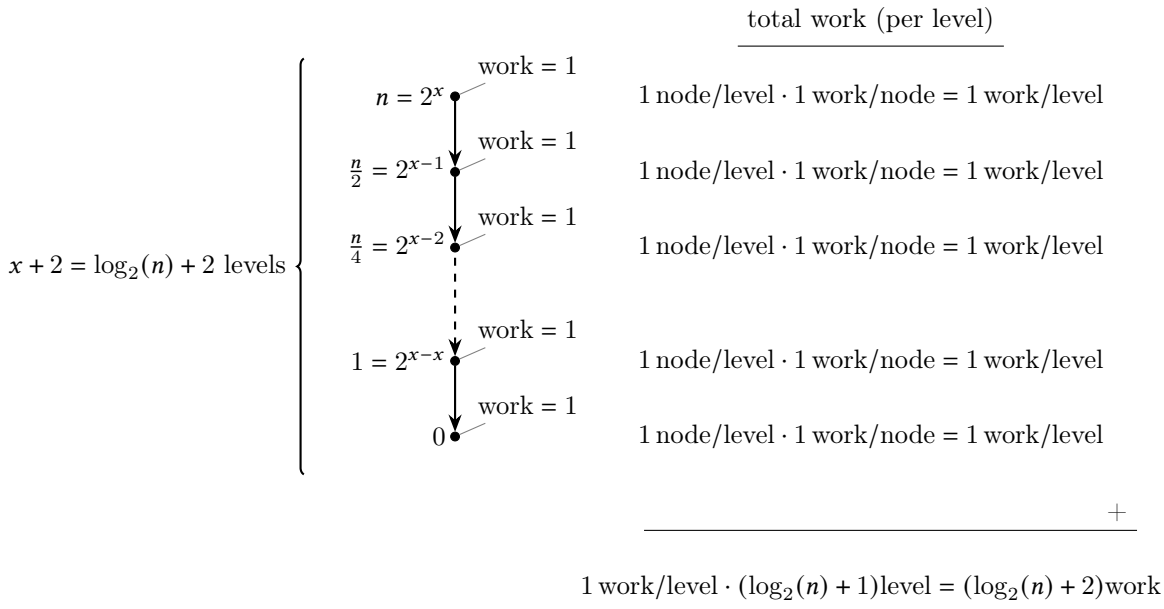


Figure 3.19: The recursion tree for LowerBoundRec together with a per-level summation of the amount of work done by LowerBoundRec when inspecting a list of $n = 2^x$ values.

1. How much work is done *per level* of the tree, this by determining:
   (a) how many nodes there are on that level of the tree, and
   (b) how much work is represented by each node at that level;
2. How many levels the tree has.

Example 3.38. Consider a call to LowerBoundRec. We will determine an *upper bound* on the amount of work done by LowerBoundRec by drawing a recursion tree. To simplify our recursion tree, we assume that the size (*end* − *begin*) of the part of list L we are inspecting is a power-of-two: $n = 2^x$ for some $x \geq 0$. We have drawn the recursion tree of LowerBoundRec together with a per-level summation of the amount of work done in Figure 3.19.

From the recursion tree, we obtain that the complexity of LowerBoundRec is modelled by $\mathsf{LBModel}(n) = \log_2(n) + 2$ if the size n of the part of the list inspected by LowerBoundRec is a power-of-two. To obtain a strict complexity bound for LowerBoundRec for *all* n, we observe that the complexity of a call to LowerBoundRec that inspects a part of a list of size n, $2^{x-1} < n \leq 2^x$, is lower bounded by $x+1 = \lceil \log_2(n) \rceil + 1$ (the complexity of a call to LowerBoundRec with size $2^{x-1} < n$) and is upper bounded by $x+2 = \lceil \log_2(n) \rceil + 2$ (the complexity of a call to LowerBoundRec with size $n \leq 2^x$). Hence, we conclude that the runtime complexity of LowerBoundRec is $\Theta(\log_2(n))$.

We can also model the memory complexity of recursive algorithms. When doing so, it is important to keep in mind that *every function call* requires some memory (e.g., to store the local variables of the that call to the function, to keep track of where the function was called, and so on).

Example 3.39. Each recursive call of LowerBoundRec uses a constant amount of memory. Hence, we conclude that the memory complexity of LowerBoundRec is modelled by the recurrence

$$T(n) = 1 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$$

and we conclude that the memory complexity of LowerBoundRec is $\Theta(\log_2(n))$.

Proposition 3.40. *The runtime and memory complexity of* LowerBoundRec *is* $\Theta(\log_2(n))$ *with* n *the size of the part of the list inspected by* LowerBoundRec.

Algorithm LowerBound(L, v, *begin*, *end*) **:**
Pre: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.
```
1: while begin ≠ end do
2:    mid := (begin + end) div 2.
3:    if L[mid] < v then
4:       begin := mid + 1.
5:    else
6:       end := mid.
7:    end if
8: end while
9: return begin.
```
Post: return the first offset r, $begin \leq r < end$, with $v \leq L[r]$ or, if no such offset exists, $r = end$.

Figure 3.20: The LowerBound algorithm that uses *binary searching* to find the first position p in L with $v \leq L[p]$.

The runtime complexity $\Theta(\log_2(n))$ of LowerBoundRec is significantly better than the worst case runtime complexity of $\Theta(|L|)$ of LinearSearch. This improved runtime complexity is enabled entirely by the restrictions placed on the input list L by LowerBoundRec.

Remark 3.41. Often, more efficient solutions to a problem can be obtained by *placing restrictions* on the input such that the input guarantees a useful structure that simplifies solving our problem.

The LowerBoundRec is a very simple recursive algorithm. In this case, we can even eliminate the recursion altogether. By doing so, we obtain the LowerBound algorithm of Figure 3.20.

Theorem 3.42. *The LowerBound algorithm of Figure 3.20 is correct. The runtime complexity of LowerBound is $\Theta(\log_2(n))$, with n the size of the part of the list inspected by LowerBound, and the memory complexity of LowerBound is $\Theta(1)$.*

In theory, the algorithms LowerBoundRec and LowerBound have the same runtime complexity. In practice, the LowerBound algorithm is preferred, however, as a while loop has less overhead than a function call (e.g., LowerBound uses less memory).

⋆ *Remark* 3.43. Some compilers can optimize away recursion when it is not necessary. Such compilers are able to automatically translate LowerBoundRec into LowerBound, in which case there is no practical difference between the two.

Exercise 3.11. Consider the algorithm LowerBoundV of Figure 3.21, a variant of the LowerBound algorithm. Is this algorithm correct? How do the best case complexity and worst case complexity of LowerBoundV compare with LowerBound? Which of these algorithms is best-suited when solving the *contains problem* of Problem 3.1 on ordered lists?

Exercise 3.12. What is the runtime complexity of the FastPower algorithm of Figure 3.7?

Exercise 3.13. Assume that the LowerBoundV algorithm of Figure 3.21 is *only used* with inputs L and v such that $v \in L$. Hence, LowerBoundV is only used to find the offset of values v that are in L. What is the *average case* complexity of LowerBoundV under this assumption?

Exercise 3.14. Assume n is an exact power of 2 ($n = 2^j$ for some natural number j) and consider the recurrence

$$T(n) = \begin{cases} 5 & \text{if } n = 1; \\ 4T\left(\frac{n}{2}\right) + 7n & \text{if } n > 1. \end{cases}$$

Use induction to prove that $T(n) = f(n)$ with $f(n) = 5n^2 + 7n(n-1)$.

Algorithm LowerBoundV(L, v, *begin*, *end*) **:**
Pre: L is an ordered *array*, v a value, and $0 \leq begin \leq end \leq |L|$.

```
 1: while begin ≠ end do
 2:    mid := (begin + end) div 2.
 3:    if L[mid] < v then
 4:       begin := mid + 1.
 5:    else if L[mid] = v then
 6:       return mid.
 7:    else
 8:       end := mid.
 9:    end if
10: end while
11: return begin.
```

Post: return an offset r, $begin \leq r < end$, with $v \leq L[r]$ or, if no such offset exists, $r = end$.

Figure 3.21: The LowerBoundV algorithm, a variant of the LowerBound algorithm, that uses *binary searching* to find any position p in L with $v \leq L[p]$.

3.3.1 ⋆ Solving Problems using Binary Search

Binary search, as discussed in Section 3.3, is typically used to search for some value v in an ordered list L. The fundamental working of binary search is based on two main principles:

1. we are operating on an ordered list L of values; and
2. inspecting a value $L[i]$, $0 \leq i < |L|$, tells us whether the outcome we are looking for is found at $L[i]$, can be found *before* the value $L[i]$ (in list $L[0, i)$), or can be found *after* the value $L[i]$ (in list $L[i, |L|)$).

These principles apply to many problems, including problems that have little in common with searching in a list. Variants of the binary search algorithm can be applied to solve such problems. Next, we look at one such a problem: the problem of finding the length of a list of *unknown length*.

Problem 3.44. Let L be list of values of unknown length and assume we have an efficient inspect-function Inspect(L, i) that returns `true` if the list has an i-th value and returns `false` otherwise. The *list-length problem* is the problem of finding the length of list L.

Example 3.45. Consider the list L = ['apple', 'pear', 'orange'] with value 'apple' as the 0-th value, 'pear' as the 1-st value, and 'orange' as the 2-nd value. Hence,

$$\text{Inspect}(L, 0) = \texttt{true} \qquad \text{Inspect}(L, 1) = \texttt{true} \qquad \text{Inspect}(L, 2) = \texttt{true} \qquad \text{Inspect}(L, 3) = \texttt{false}.$$

The list L has length 3, which is reflected by the fact that $i = 3$ is the smallest value for which Inspect(L, i) is `false`.

The algorithm ListLength of Figure 3.22 applies the *linear search*-style strategy of the above example to solve the list-length problem of Problem 3.44.

Proposition 3.46. *The ListLength algorithm is correct and has a runtime complexity of* $\Theta(|L|)$.

Next, we will look at solving the list-length problem more efficiently using the principles of binary search. To do so, we first identify how these principles apply to the list-length problem. We observe that:

1. we are operating on an ordered list $0, 1, \ldots$ of possible lengths of lists;
2. inspecting a list length i using Inspect(L, i) tells us whether the list has length at-most i (Inspect(L, i) = `false`) or whether the list has a length of more than i (Inspect(L, i) = `true`).

Algorithm ListLength(L) :
Pre: L is an *array* of unknown length.
1: $len := 0$.
2: **while** Inspect(L, len) **do**
3: $len := len + 1$.
4: **end while**
5: **return** len.
Post: return the length $|L|$ of array L.

Figure 3.22: The ListLength algorithm.

Algorithm LBListLength(L, N) :
Pre: L is an *array* of unknown length $|L| \leq N$.
1: $begin, end := 0, N$.
2: **while** $begin \neq end$ **do**
3: $mid := (begin + end) \operatorname{div} 2$.
4: **if** Inspect(L, mid) **then**
5: $begin := mid + 1$.
6: **else**
7: $end := mid$.
8: **end if**
9: **end while**
10: **return** $begin$.
Post: return the length $|L|$ of array L.

Algorithm ListLengthUB(L) :
Pre: L is an *array* of unknown length.
11: $n := 1$.
12: **while** Inspect(L, n) **do**
13: $n := 2 \cdot n$.
14: **end while**
15: **return** n.
Post: return N, $|L| \leq N = 1$ or $|L| \leq N < 2|L|$.

Figure 3.23: The LBListLength algorithm to find the length of a list L using binary search. The LBListLength algorithm requires an upper bound N, $|L| \leq N$, on the length of list L. The ListLengthUB algorithm can be used to efficiently compute such an upper bound N with $|L| \leq N < 2|L|$.

These observations indicate that a variant of binary search can be used.

We cannot simply use the above observations: we are searching a *potential* list-length in an *unbounded* list of lengths $0, 1, \ldots$, whereas binary search requires an end-point for the search. For now, we assume that we can efficiently derive an upper bound N, $|L| \leq N \leq 2|L|$, on the length of list L. Using the upper bound N and the above observations, we derive the LBListLength algorithm of Figure 3.23, a variant of the LowerBound algorithm, to solve the list-length problem.

We cannot simply use the LBListLength algorithm, however: we also need to determine the upper bound N efficiently. To compute N, we can use the ListLengthUB algorithm, which uses *exponential backoff* to compute N in a few steps. We have:

Theorem 3.47. *LBListLength(L, ListLengthUB(L)) computes the length of list L and has a runtime complexity of* $\Theta(\log_2(|L|))$.

Exercise 3.15. Prove that the LBListLength and ListLengthUB algorithms of Figure 3.23 are correct.

Algorithm RangeQuery$(L, [v, w])$ **:**
Pre: L is an ordered *array*, v, w are values, and $v \leq w$.
1: $i :=$ LowerBound$(L, v, 0, |L|)$.
2: $j := i$.
3: **while** $j \leq |L|$ **and also** $L[j] \leq w$ **do**
4: $\quad j := j + 1$.
5: **end while**
6: **return** $L[i, j)$.
Post: return the list $L[m, n)$, $0 \leq m \leq n \leq |L|$, such that $L[m, n)$ is the list of all values $e \in L$ with $v \leq e \leq w$.

Figure 3.24: The RangeQuery algorithm.

Exercise 3.16. Argue that the ListLengthUB algorithm of Figure 3.23 has a runtime complexity of $\Theta(\log_2(|L|))$. Assuming that ListLengthUB has a runtime complexity of $\Theta(\log_2(|L|))$, argue that LB-ListLength$(L,$ ListLengthUB$(L))$ has a runtime complexity of $\Theta(\log_2(|L|))$.

3.3.2 Range Queries

Binary search already showed that the assumption that a list is *ordered* can speed up basic list operations significantly: LowerBound can find the first occurrence of a value v in an ordered list L in $\Theta(\log_2(|L|))$, whereas LinearSearch will do so in worst case $\Theta(|L|)$.

As the assumption that data is *ordered* is very powerful, this assumption is often explicitly maintained by software that manages large collections of data (e.g., database management systems). As we have argued before, software can even maintain an order for free for many forms of data:

Example 3.48. Consider the relation enrolled$(dept, code, sid, date)$ that models a list of all students (identified by the student identifier *sid*) enrolled for a course (identified by the department *dept* and course code *code*) together with the enrollment date of the student for the course (identified by *date*).

If we add new enrollment data to enrolled by always adding newly enrolled students to the end of the list, then this relation of enrolled students is automatically ordered on *increasing enrollment date*.

We can use the LowerBound algorithm not only for finding a specific value, but also for answering range queries.

Problem 3.49. Let L be a list and $[v, w]$ be a range query with $v \leq w$. The solution of the *range query problem* for L and $[v, w]$ is the list of all values $e \in L$ with $v \leq e \leq w$.

The RangeQuery algorithm of Figure 3.24 solves the range query problem of Problem 3.49.

Example 3.50. Assume we have the relation enrolled$(dept, code, sid, date)$ of Example 3.48. This relation models a list of enrollment data ordered by the enrollment date of the student. To obtain the list of all students enrolled for a course in 2023, we simply call

$$\text{RangeQuery}(\text{enrolled}, [(\text{`'}, \text{`'}, -1, 2023), (\text{`'}, \text{`'}, -1, 2024)]),$$

in which the empty strings `` is used as a *minimum value* for the attributes *dept* and *code* and -1 as a *value* smaller than any valid student identifier (attribute *sid*).

Next, we will consider the runtime complexity of the RangeQuery algorithm. At Line 1, we call the LowerBound algorithm. By Theorem 3.42, the runtime complexity of this call is $\Theta(\log_2(|L|))$. Next, the while loop at Line 3 will, in the worst case, inspect every value $L[i]$, $0 \leq i \leq |L|$. Hence, the worst case complexity of the while loop of RangeQuery is $\Theta(|L|)$ and we conclude that the worst case complexity of the RangeQuery algorithm is $\Theta(\log_2(|L|) + |L|) = \Theta(|L|)$. This is a discouraging result: in the worst case, RangeQuery is as efficient as a simple loop that checks *every* value in list L!

Luckily the runtime of the RangeQuery algorithm will *only* see the worst case complexity if *all values* in list L are returned. Indeed, RangeQuery has a best case complexity of $\Theta(\log_2(|L|))$ whenever *no values* are returned. Hence, the best case and worst case complexity of RangeQuery are rather different.

As with the LinearSearch algorithm studied in Section 3.2, the difference between best case and worst case complexity of RangeQuery are indicative of a limitation of our runtime model: we are modeling the runtime of RangeQuery only in terms of the size of list L. Close inspection of RangeQuery shows that the size $|result|$ of the query result computed by RangeQuery also plays a significant role: the while loop at Line 3 inspects *exactly* $|result| + 1$ consecutive values in L. Hence, the *all case* complexity of RangeQuery is $\Theta(\log_2(|L|) + |result|)$.

Theorem 3.51. *The RangeQuery algorithm of Figure 3.24 is correct. The runtime complexity of RangeQuery is* $\Theta(\log_2(|L|) + |result|)$ *and the memory complexity of RangeQuery is* $\Theta(1)$.

Exercise 3.17. Prove that the RangeQuery algorithms of Figure 3.24 is correct.

3.3.3 Optimizing Joins using Range Queries

A join of two-or-more lists $L_1, \ldots, L_n$ of values computes a single new list in which each list value is computed from a combination of values $v_1 \in L_1, \ldots, v_n \in L_n$ according to some *join condition*.

Consider the relation products($name, category$) that models a product listing in which each product has a name *name* and main category *category* and the relation categories($category, related$) that relates category *category* to a related category *related*. We refer to Figure 3.25, *left*, for an example of these relations.

Next, we shall look at the *join* of products and categories that computes the list of pairs ($name, related$) that relates each product *name* with the categories the product is related to. We refer to Figure 3.25, *right*, for an example of the result of this join. A simple *nested-loop* algorithm can compute this join, e.g., the NestedLoopPC algorithm of Figure 3.26.

The NestedLoopPC algorithm is obviously correct. To determine the complexity of NestedLoopPC, we notice that every combination of a product listing in products and a related category listing in categories is inspected exactly once. Specifically, each category listing in categories is inspected *once* during the for loop at Line 3. The for loop at Line 3 is executed *once* for every product listing that is inspected by the for loop at Line 2. Hence, we conclude:

Proposition 3.52. *The NestedLoopPC algorithm is correct and has a runtime complexity of* $\Theta(|product| \cdot |categories|)$.

Given a product listing $(p.n, p.c) \in$ products, the for loop at Line 3 of the NestedLoopPC algorithm will visit every category listing in the categories relation, even if the category $p.c$ of the product listing under consideration is only related to a few other categories. As one can expect that each category is only related to a handful of other categories, this means that NestedLoopPC is *inefficient*.

As the categories do not change often in practice, one can choose to maintain the relation categories in an ordered manner (even though this increases the cost of adding category listings to categories). If categories is ordered, then we can use LowerBound to search in categories only those category listings related to the category $p.c$ instead of inspecting every category listing. By incorporating these changes, we obtain the NestedBinaryPC algorithm of Figure 3.27.

We note that Lines 3–7 essentially perform a range query that searches for all categories in categories related to the category $p.c$ (hence, all rows of the form $(p.c, ?)$). Hence, using the same argument as used in

products

name	*category*
Apple	Fruit
Bok choy	Vegetable
Canelé	Pastry
Donut	Pastry

categories

category	*related*
Fruit	Food
Fruit	Produce
Pastry	Food
Vegetable	Food
Vegetable	Produce

Join Result

name	*related*
Apple	Food
Apple	Produce
Bok choy	Food
Bok choy	Produce
Canelé	Food
Donut	Food

Figure 3.25: An example of the relations products and category on the *left* and an example of a relation derived from the combination of these relations on the *right*.

Algorithm NestedLoopPC(products, categories) **:**
Pre: relations products($name, category$) and categories($category, related$).
1: $output := \emptyset$.
2: **for** $(p.n, p.c) \in$ products **do**
3: **for** $(c.c, c.r) \in$ categories **do**
4: **if** $p.c = c.c$ **then**
5: add $(p.n, c.r)$ to $output$.
6: **end if**
7: **end for**
8: **end for**
Post: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \wedge ((c.c, c.r) \in \text{categories})\}$.

Figure 3.26: Computing a join of products and categories using a nested-loop algorithm.

our analysis of the RangeQuery algorithm (see Section 3.3.2), we conclude that the complexity of Lines 3–7 is $\Theta(\log_2(|categories|) + w)$ with w the number of values written to *output* at Line 5. We conclude:

Theorem 3.53. *The NestedBinaryPC algorithm is correct and has a runtime complexity of* $\Theta(|product| \cdot \log_2(|categories|) + |result|)$.

3.4 Chapter Notes

Most standard programming libraries, e.g., for C++ and Java, provide ready-to-use implementations of the search algorithms presented in this chapter. We refer to the overview in Figure 3.28.

Algorithm NestedBinaryPC(products, categories) **:**
Pre: relations products($name, category$) and categories($category, related$), relation categories ordered.
1: $output := \emptyset$.
2: **for** $(p.n, p.c) \in$ products **do**
3: $i :=$ LowerBound(categories, $(p.c, \text{''}), 0, |\text{categories}|)$.
4: **while** $i < |\text{categories}|$ **and also** categories$[i].category = p.c$ **do**
5: add $(p.n, \text{categories}[i].related)$ to $output$.
6: $i := i + 1$.
7: **end while**
8: **end for**
Post: return $\{(p.n, p.c) \mid ((p.n, p.c) \in \text{products}) \wedge ((c.c, c.r) \in \text{categories})\}$.

Figure 3.27: Computing a join of products and categories using a binary search to find related categories.

Algorithm	C++	Java
Contains	`std::ranges::contains`	*collection*`.contains`[a]
LinearSearch	`std::find`	*collection*`.indexOf`[a]
LinearPredSearch	`std::find_if`	`java.util.stream::filter`[b]
LowerBound	`std::lower_bound` `std::upper_bound`[d]	`java.util.Arrays::binarySearch`[c]
Related libraries	`<algorithm>`, `<ranges>`	`java.util.Arrays`, `java.util.ArrayList`, ...

[a] Here, *collection* is a standard Java data collection such as `java.util.ArrayList`.
[b] Using the stream library supported by standard Java data collections.
[c] Does not guarantee to return the offset of the *first* occurrence of a value.
[d] Returns the offset of the first element in the list that is strictly larger than the searched-for value.

Figure 3.28: Overview of existing implementations of the main algorithms studied in this chapter in the standard programming libraries of C++ and Java.