"On two occasions I have been asked, 'If you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

– Charles Babbage (1791-1871)
Inventor of the first mechanical computer

"In theory, theory and practice are the same. In practice, they are not."

– Yogi Berra

Lecture I
OUTLINE OF ALGORITHMICS

This first chapter is mostly informal. The rest of this book has no dependency on this chapter, save the definitions in §7 concerning asymptotic notations. Hence a light reading may be sufficient. We recommend re-reading this chapter after finishing the rest of the book, when many of the remarks here may take on more concrete meaning.

An appendix collects useful mathematical concepts that are used throughout the book.

In computer science, we study problems that can be solved on computers. Such problems can be roughly classified into problems-in-the-large and problems-in-the-small. The former is associated with large software systems such as an airline reservation system, compilers or text editors. The latter[1] is identified with mathematically well-defined problems such as sorting, multiplying two matrices or solving a linear program. The methodology for studying such "large" and "small" problems are quite distinct: Algorithmics is the study of the small problems and their algorithmic solution. In this introductory lecture, we present an outline of this enterprise.

Algorithmics is about "small" problems

Throughout this book, **computational problems** (or simply "problems") refer to problems-in-the-small. It is the only kind of problem we address. We assume the student is familiar with computer programming and has a course in data structures and some background in discrete mathematics.

¶1. Book Organization. The chapters in this book are numbered by capital roman numerals: I, II, III, IV, etc. The chapters are organized into sections, denoted §1, §2, §3, etc. Occasionally, we have subsections such as §3.1, §3.2, etc. Independent of the sections and subsections, a chapter is also organized into **numbered paragraphs**, denoted ¶1, ¶2, ¶3, etc. Optional or advanced sections or paragraphs are marked by an asterisk, as in §*2 or ¶*37.

[1] If problems-in-the-large is macro-economics, then the problems-in-the-small is micro-economics.

When referring to sections or paragraphs across in another chapter, we write, e.g., §*III*.8 (section 8 in Chapter III) or ¶*IX*.3 (paragraph 3 in Chapter IX). Note we use[2] colored fonts; hyperlinks are available in the pdf file.

§1. What is Algorithmics?

Algorithmics is the systematic study of efficient algorithms for computational problems; it includes techniques of algorithm design, data structures, and mathematical tools for analyzing algorithms.

Why is algorithmics important? Because algorithms is at the core of all applications of computers. These algorithms are the "computational engines" that drive larger software systems. Hence it is important to learn how to construct algorithms and to analyze them. Although algorithmics provide the building blocks for large application systems, the construction of such systems usually require additional non-algorithmic techniques (e.g., database theory) which are outside our scope.

¶2. We can classify algorithmics according to its applications in subfields of the sciences and mathematics: thus we have computational geometry, computational topology, computational number theory, computer algebra, computational statistics, computational finance, computational physics, and computational biology, etc. More generally, we have "computational X" where X can be any discipline. But another way to classify algorithmics is to look at the generic tools and techniques that are largely independent any discipline. Thus, we have sorting techniques, graph searching, string algorithms, dynamic programming, numerical techniques, etc, that cut across individual disciplines. Thus we have identified two dimensions along which the field of algorithmics can be classified. Let us represent these two orthogonal classification schemes using a matrix:

	geometry	topology	finance	physics	biology	algebra	...
sorting	✓	✓	✓	✓	✓	✓	
graph searching	✓		✓		✓		
string algorithms		✓		✓	✓		
dynamic programming	✓				✓	✓	
numerical methods			✓	✓			
⋮	⋮						

Computer Science is row-oriented

So each computational X is represented by a column in this matrix, and each computational technique is represented by a row. Each check mark indicates that a particular computational technique is used in a particular discipline X. Individual scientific disciplines take a column-oriented view, but *Computer Science (and also this book) takes the row-oriented view of this matrix*. In other words, we study the common elements that characterize each row. These row labels can be grouped into four basic themes:

(a) data-structures (e.g, linked lists, stacks, search trees)

[2] Students may request a no-color version.

(b) algorithmic techniques (e.g., divide-and-conquer, dynamic programming)

(c) basic computational problems (e.g., sorting, graph-search, point location)

(d) analysis techniques (e.g., recurrences, amortization, randomized analysis)

These themes interplay with each other. For instance, some data-structures naturally suggest certain algorithmic techniques (e.g., graphs requires graph-search techniques). Or, an algorithmic technique may entail certain analysis methods (e.g., divide-and-conquer algorithms require recurrence solving). The field of complexity theory in computer science provides some unifying concepts for algorithmics; but complexity theory is too abstract to capture many finer distinctions we wish to make. Thus algorithmics often makes domain-dependent assumptions. For example, in the subfield of computer algebra, the complexity model takes each algebraic operation as a primitive, while in the subfield of computational number theory, these algebraic operations are reduced to some bit-complexity model primitives. In this sense, algorithmics is more like combinatorics (which is eclectic) than group theory (which has a unified framework). Students may initially find this eclectic nature of algorithmics confusing. But ultimately, we hope the student will develop an "algorithmic frame of mind" that sees an over-arching unity in this jumble of topics.

§2. Computational Problems: What are we solving?

Despite its name, the starting point for algorithmics is not algorithms, but **computational problems**. But what are "computational problems"? We mention three main categories.

¶3. (A) Input-output problems. Such problems are the simplest to understand. A **computational problem** is a precise specification of input-and-output (I/O) formats, and for each input instance I, we have a set of possible output instances $\mathcal{O}(I)$. The word "formats" emphasizes the fact the representation of the input I and outputs $\mathcal{O}(I)$ are part and parcel of the problem. Inputs that are improperly formatted are meaningless. In practice, standard representations may be taken for granted (e.g., numbers are assumed to be in binary and set elements are arbitrarily listed without repetition). Note that the input-output relationship need not be functional: a given input may have several acceptable outputs when $|\mathcal{O}(I)| > 1$.

Cf. the opening quote of Babbage hints at I/O problems?

(A1) Sorting Problem. The input is a sequence of numbers $(a_1, \ldots, a_n)$ and output is a rearrangement of these numbers $(a'_1, \ldots, a'_n)$ in non-decreasing order. An input instance is $(2, 5, 2, 1, 7)$, with corresponding output instance $(1, 2, 2, 5, 7)$.

(A2) Primality Testing. Input is a natural number n and output is either YES (if n is prime) or NO (if n is composite). Numbers are assumed to be encoded in decimal. E.g., if the input is 123 then the output is NO. But for the input 23, the output is YES. This is an example of a **decision or recognition problem**, where the output have only two possible answers (YES/NO, 0/1, Accept/Reject).

Simplest imaginable type of problem?

One can generalize this to problems whose output comes from a finite set. For instance, in computational geometry, the decision problems tend to have three possible answers: Positive/Negative/Zero or IN/OUT/ON. For instance, the **point classification problem** is where we are given a point and some geometric object such as a triangle or a cell. The point is either inside the cell, outside the cell or on the boundary of the cell.

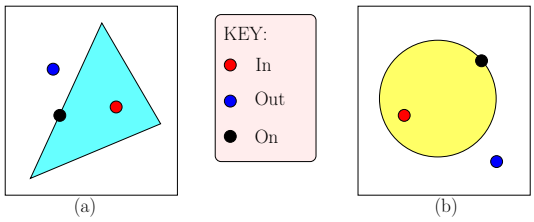


Figure 1: Classifying points relative to (a) Triangles, (b) Discs

¶4. (B) Preprocessing problems. A generalization of input-output problems is what we call **preprocessing problem**: *given a set S of objects, construct a data structure $D(S)$ such that for an arbitrary 'query' (of a suitable type) about S, we can use $D(S)$ to efficiently answer the query.* There are two distinct stages in such problems: preprocessing stage and a "run-time" stage. Usually, the set S is "static" meaning that membership in S does not change under querying.

Two-staged problems

(B1) Ranking Problem. The preprocessing input is a set S of numbers. A query on S is a number q for which we like to determine its rank in S. The rank of q in S is the number of items in S that are smaller than or equal to q. E.g., if $S = \{3, 2, 7, 5\}$ then the rank of $q = 6$ in S is 3. If $|S| = n$, we can let $D(S)$ be a sorted array $A[1..n]$ containing the elements of S in non-decreasing order. Then using a binary search, the rank of q is equal to the unique i such that $A[i] \leq q < A[i+1]$ (with $A[0] = -\infty$). Thus the preprocessing amounts to sorting the set S, and querying is binary search of an array.

(B2) Post Office Problem. Many problems in computational geometry and database search are the preprocessing type. The following is a geometric-database illustration: given a set S of points in the plane, find a data structure $D(S)$ such that for any query point p, we find an element in S that is closest to p. (Think of S as a set of post offices and we want to know the nearest post office to any position p). Note that the 1-dimensional version of this problem is closely allied to the ranking problem.

Two algorithms are needed to solve a preprocessing problem: one to construct $D(S)$ and another to answer queries. They correspond to the two stages of computation: an initial **preprocessing stage** to construct $D(S)$, and a subsequent **querying stage** in which the data structure $D(S)$ is used. There may be a tradeoff between the **preprocessing complexity** and the **query complexity**: $D_1(S)$ may be faster to construct than an alternative $D_2(S)$, but answering queries using $D_1(S)$ may be less efficient than $D_2(S)$. But our general attitude is to prefer $D_2(S)$ over $D_1(S)$ in this case: we prefer data structures $D(S)$ that support the fastest possible query complexity. Our attitude is often justified because the preprocessing complexity is a one-time cost, but query complexity is a recurring cost.

Preprocessing problems are a special case of **partial evaluation problems**. In such problems, we construct partial answers or intermediate structures based on part of the inputs; these partial answers or intermediate structures must somehow anticipate all possible extensions of the partial inputs.

¶* 5. (C) Dynamization and Online problems. Now assume the input S is a set of objects. For example, a database might be regarded as a set. If S can be modified under queries, then we have a **dynamization problem**: with S and $D(S)$ as above, we must now design our solution with an eye to the possibility of modifying S (and hence $D(S)$). Typically,

we want to insert and delete elements in S while at the same time answering queries on $D(S)$ as before. A set S whose members can vary over time is called a **dynamic set** and hence the name for this class of problems.

Here is another formulation: *we are given a sequence* $(r_1, r_2, \ldots, r_n)$ *of* ***requests****, where a request is one of two types: either an* ***update*** *or a* ***query****. We want to 'preprocess' the requests in an online fashion, while maintaining a time-varying data structure* D*: for each update request, we modify* D *and for each query request, we use* D *to compute and retrieve an answer (*D *may be modified as a result).*

In the simplest case, updates are either "insert an object" or "delete an object" while queries are "is object x in S?". This is sometimes called the **set maintenance problem**. The preprocessing problems can be viewed as a set maintenance problem in which we first process a sequence of insertions (to build up the set S), followed by a sequence of queries.

(C1) Dynamic Ranking Problem. Any preprocessing problem can be systematically converted into a set maintenance problem. For instance, the ranking problem (B1) turns into the **dynamic ranking problem** in which we dynamically maintain the set S subject to intermittent rank queries. Recall that in the static problem (B1) above, the data structure $D(S)$ is just a sorted array. This will not be very efficient under insertion and deletion. The typical data structure here is some form of **dynamic search trees** (see in Chapter 3).

(C2) Graph Maintenance Problems. Dynamization problems on graphs are more complicated than set maintenance problems (though one can still view it as maintaining a set of edges). One such problem is the **dynamic connected component problem**: updates are insertion or deletion of edges and/or vertices. Queries are pairs of vertices in the current graph, and we want to know if they are in the same component. The graphs can be directed or undirected.

¶6. (D) Pseudo-problems. Let us illustrate what we regard to be a pseudo-problem from the viewpoint of our subject. Suppose your boss asks your IT department to "build an integrated accounting system-cum-employee database". This may be a real world scenario but it is not a legitimate topic for algorithmics because part of the task is to figure out what the input and output of the system should be, and there are probably other implicit non-quantifiable criteria (such as available technology and economic realities).

§3. Computational Model: How do we solve problems?

Once we agree on the computational problem to be solved, we must choose the tools for solving it. This is given by the **computational model**. Any conventional programming language such as `C` or `Java` (suitably abstracted, so that it does not have finite space bounds, etc) can be regarded as a computational model. A computational model is specified by

(a) the kind of data objects that it deals with

(b) the primitive operations to operate on these objects

(c) rules for composing primitive operations into larger units called **programs**.

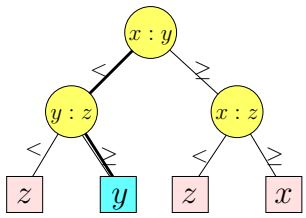
Programs can be viewed as individual instances of a computational model. For instance, the

Turing model of computation is an important model in complexity theory and the programs here are called Turing machines.

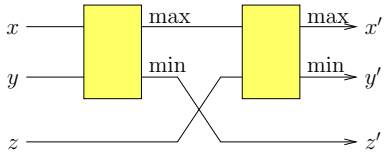
¶7. Models for Comparison-based Problems. The sorting problem has been extensively studied since the beginning of Computer Science (from the 1950's). Sorting is just a representative of a whole class of problems that can be solved using the primitive capability of comparing two elements. It turns out that there are several distinct computational models for such problems. We will next describe three of them: the **comparison tree model**, the **comparator circuit model**, and the **tape model**. In each model, the data objects are elements from a linear order.

3 sorting models

First we consider the comparison tree model which has only one primitive operation, viz., comparing the two elements x, y resulting in one of two outcomes $x < y$ or $x \geq y$. Such a comparison is usually denoted "$x : y$". We compose these primitive comparisons into a **tree program** by putting them at the internal nodes of binary tree. Tree programs only represent flow of control and are, more generally, are called **decision trees** where the decision can be based on predicates other than comparisons. Figure 2(a) illustrates a comparison tree on inputs x, y, z.



(a) Comparison Tree



(b) Comparator Circuit

Figure 2: Two programs to find the maximum of x, y, z.

To use a comparison tree, we begin at the root, and perform the indicated comparison, say $x : y$. If $x < y$, we proceed to the left child; otherwise, we proceed to the right child. We continue recursively in this manner until we reach a leaf, and stop. Let us illustrate this with the comparison tree in Figure 2(a) (follow the thick path from root to a leaf). Suppose our input is $\{x = 7, y = 9, z = 3\}$. Then the comparison $x : y$ at the root tells us to compare $7 : 9$. Since $7 < 9$, we move to the left child. The comparison at the left child is $y : z$, i.e., $9 : 3$. Since $9 \geq 3$, we move to the right child. We have reached a leaf. This leaf specifies the element y (i.e., 9) which is our output. The reason for this output is that our comparison tree is supposed to be an algorithm to find a maximum of x, y, z.

So the outputs of a comparison tree are specified at each leaf. For instance, if the tree is meant for sorting, each leaf will output the sorted order of the input set. These examples raise the question: what exactly is the nature of the output at each leaf? There is a precise answer: the output at each leaf must be determined from the set of relations collected along the edges of the path to the leaf. Let us unroll this remark: each edge of the comparison tree represents a relationship of the form $x < y$ or $x \geq y$, and the set of all these relationships along a path to a leaf v forms a partial order $\phi(v)$ on the input set. *The tree program solves problem P if for each leaf v, the partial order $\phi(v)$ determines a solution to P.* Adopting a notation from mathematical logic, we may write "$\phi(v) \vDash P$" to indicate this relationship,

For the comparison tree Figure 2(a), you may verify that the path taken by the input

$\{x = 7, y = 9, z = 3\}$ collected the partial order $\phi(v) = \{x < y, y \geq z\}$ which does determine y (the output) as a maximum. The output at the remaining three leaves of the tree is likewise verified.

We come to the second computational model for sorting-like problems: we again have only one kind of primitive called a **comparator** that takes two input elements x, y and returns two outputs, $\max\{x, y\}$ and $\min\{x, y\}$. Each comparator can be viewed as a node in a graph with two incoming edges and two outgoing edges. Graphical conventions: *inputs enter from the left and exit to the right of each comparator; the maximum output is drawn above the minimum output.* Comparators are composed into **(comparator) circuits** as illustrated in Figure 2(b). This circuit has inputs x, y, z and outputs x', y', z' and two comparators (shown as rectangles). For instance, the output x' in the circuit Figure 2(b) is the maximum of x, y, z; if the circuit is regarded as an algorithm for computing the maxima, then we can ignore the outputs y', z'. For the sorting problem, no output may be ignored.

Formally, a **comparator graph** is a directed acyclic graph (DAG) whose nodes are classified as input, output or comparator nodes. Each type of node is characterized by their (in,out)-degrees: $(0, 1)$ for input, $(1, 0)$ for output, and $(2, 2)$ for comparators. It follows that the number of input and output nodes are equal (proof?). The edges are called **wires**. Note that a comparator graph has captured all the essentials of a comparator circuit except one: to make the graph equivalent to a circuit, we need to "order" the two output wires from each comparator node: we must designate one of the output wires as "max" and the other as "min".

What is the semantics? The inputs are to be instantiated with elements from any totally ordered set (say, real numbers). If the input nodes are labeled $x_1, \ldots, x_n$, we can label each wire with a min-max expression over $x_1, \ldots, x_n$. If the two input wires to a comparator has the expression E_1, E_2 then the output wires are labeled $\min(E, E')$ and $\max(E, E')$, consistent with the above min/max output designations. For instance, in Figure 2(b), the labels x', y', z' can be interpreted as these min-max expressions:

$$x' = \max(\max(x, y), z), y' = \min(\max(x, y), z), z' = \min(x, y).$$

In general, if the output nodes of the circuit are labeled $x'_1, \ldots, x'_n$, then it is clear that $(x'_1, \ldots, x'_n)$ represents a permutation of $(x_1, \ldots, x_n)$. Thus comparator circuits compute permutations of the input.

The third model for sorting is the tape model, studied in [3]. In this model, we assume a fixed number of sequential tapes, where each tape can store a sequence of items. At any moment, each tape has a head position. We can read the item in this head position, or we can write an item into this head position. We call this read/write operation an **advance operation**. There is one other operation on a tape, called the **reset operation**. This puts the tape head in the initial position. We can also test if the head position has moved past the last item on the tape. Below, we give a concrete example on how to use these tape operations. It is important to understand that the advance operation is cheap, but a reset operation is expensive. Reset is expensive because to get to the beginning of a physical tape, you need to unwind the entire tape on to another initially empty spool.

One model of how tapes work is the audio cassette tape (remember those?). Each end of the cassette tape is attached to a separate spool, and the two spools are positioned at a fixed distance apart. The tape is wound up on each spool so that "free tape" between the two spools is held taut: the head position is placed somewhere on the free tape. Unwinding the tape on one spool requires a corresponding winding up on the other spool to keep this free tape taut. To "reset" means to completely unwind one of the spools. Unlike a cassette tape, a computer tape is wound up on one spool only. So to read/write on such a tape, we need to first physically attach the free end of the computer tape to another spool, and operate it like a cassette tape.

The tape model was important in the early days of computing when main memory was expensive and physical tapes were the standard medium for storing large data sets. Interestingly, with the advent of the web-age, a variant of this model called **streaming data model** is coming back. Now we are faced with huge amounts of real time data, and instead of sorting, we often need to compute some function of the data. Because of the large volume of data, we do not want to store this information but allow only one pass over the data. For more information about the tape model, we refer to Knuth [3, Chap. 5.4], under external sorting.

¶8. From Programs to Algorithms. We start with a problem P that we want to solve. Let A be a program in a model M. To use A to solve P, we must make sure there is a match between the data objects in the problem specification and the data objects handled by model M. If not, we can often specify some suitable encoding of the former objects by the latter. After making such encoding conventions, we may call A an **algorithm for** P if, for each legal input of P, the program A indeed computes a correct output. Thus the term "algorithm" is a semantic concept, signifying a program A in its relation to some problem P. We might indicate this relation as "$A \vDash P$", extending the notation $\phi(v) \vDash P$ above. The program A itself is a purely syntactic object, capable of more than one interpretation. E.g., the two programs in figure 2(a,b) are interpreted as algorithms to compute the maximum of x, y, z; but it is also possible to view them as algorithms for other problems (see Exercise).

¶9. The Merging Problem. A subproblem that arises in sorting is the **merge problem** where we are given two sorted lists $(x_1, x_2, \ldots, x_m)$ and $(y_1, y_2, \ldots, y_n)$ and we want to produce a sorted list $(z_1, z_2, \ldots, z_{m+n})$ where $\{z_1, \ldots, z_{m+n}\} = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$. Assume these sorted lists are non-decreasing.

The algorithmic idea for merging is as follows: what should the first output element be? Well, it is the minimum of x_1 and y_1, decided by one comparison. Assume this output is x_1. What is the next one? Well, it must be either x_2 or y_1, and another comparison will decide. So the general picture is that, for some $i, j \geq 1$, we have already output $x_1, \ldots, x_{i-1}$, and we have output $y_1, \ldots, y_{j-1}$. The next output element is either x_i or y_j, as is determined by a comparison, $x_i : y_j$. This invariant is easy to maintain. When one list is exhausted, we simply output the remaining elements in the other list. Here then is our algorithm, written in a non-specific pseudo-programming language:

Merge Algorithm
Input: $(x_1, \ldots, x_m)$ and $(y_1, \ldots, y_n)$, sorted in non-decreasing order.
Output: The merger $(z_1, \ldots, z_{m+n})$ of these two lists, in non-decreasing order.

```
▷ Initialize:
    i ← 1, j ← 1, k ← 1.
▷ Loop:
    While (i ≤ m ∧ j ≤ n)
        If (x_i < y_j)
            z_k ← x_i, i++, k++.
        else
            z_k ← y_j, j++, k++.
▷ Terminate:
    If (i > m)    ◁ The x's are exhausted, output the remaining y's
        (z_k, ..., z_{m+n}) ← (y_j, ..., y_n).
    else  ◁ The y's are exhausted, output the remaining x's
        (z_k, ..., z_{m+n}) ← (x_i, ..., x_m).
```

what is pseudo-programming?

The student should note the conventions used in our programs, such as illustrated here. In nutshell, *a pseudo-program provides a clear description of the flow of control of the algorithm, without constraining the way the operations in non-control steps of the algorithm are described.* Typically, it means we explain these operations in English or mathematical terms. Since the flow of control (loops, branches, termination) must be explicit, the pseudo-program should carefully specify how control variables such as Boolean flags or loop counter variables are modified. If we iterate over elements in some queue, we need to specify how the queue is initialized, modified or tested. Of course, computers are not smart enough to compile our programs. That is alright because our programs are intended for human consumption, not computers.

Here are some guidelines for pseudo-programs. First, we prefer English and mathematical notations over programming notations because the former are both more compact and more flexible. Natural languages (and English in particular) are highly effective for communication. Mathematics is perhaps more compact but certainly more precise. In contrast, programming notations are optimized for compilers and machines. In mathematics, "=" is a predicate, so we use "←" for assignment. Second, we use indentation for program blocks – this reduces clutter, improves readability. Third, like scripting languages, we do not declare our variables. The text and context should tell you how to interpret these variables. Finally, we use two kinds of comments: (▷ *forward comments*) to describe what is coming up next, and (◁ *backward comments*) to briefly explain the immediately preceding code (usually on the same line).

$x \leftarrow 1$, not $x = 1$.
$x \leq y$, not $x <= y$.
etc.

¶* 10. Uniform versus Non-uniform Models. The preceding merge algorithm should look more familiar to students than our comparison trees. Formally, we can regard this program as belonging to the RAM (Random Access Machine) model. It is described in Appendix B, but for now the student may just identify a RAM program with program in any conventional programming language like `Java` or `C++`. Besides the familiarity factor, there is fundamental difference between the RAM model and the comparison tree model: the former is a **uniform model** and the latter is a **non-uniform model**.

Before we explain this uniform/non-uniform distinction, let us see how we can extract from the merge algorithm of ¶9 an infinite set

$$T = \{T_{m,n} : m, n \in \mathbb{N}\} \tag{1}$$

of comparison trees. Each $T_{m,n} \in T$ is a comparison tree on the sorted input sequences $(x_1, \ldots, x_m)$ and $(y_1, \ldots, y_n)$. The root of $T_{m,n}$ has the comparison $x_1 : y_1$ because the Merge Algorithm begins with this comparison. If $x_1 < y_1$, then the Merge Algorithm will next compare $x_2 : y_1$. So we install $x_2 : y_1$ at the left child of the root. But if $x_1 \geq y_1$, the Merge Algorithm will next compare $x_1 : y_2$. Accordingly, we install $x_1 : y_2$ at the right child of the root. We can proceed this way to install a comparison at each of the node of an ever expanding comparison tree. But when there are no more comparisons, we have reached a leaf. We could install the sorted output $(z_1, \ldots, z_{m+n})$ at the leaf if we like (but formally, it is not necessary). Each $T_{m,n}$ is an algorithm for merging the sorted list $(x_1, \ldots, x_m)$ with the sorted list $(y_1, \ldots, y_n)$. The set T is called a **non-uniform algorithm** for merging. This process of constructing $T_{m,n}$ is known as "unrolling" the Merge Algorithm (for the indicated inputs). In the Exercise, we ask you to explicitly construct $T_{2,4}$ by this unrolling process.

unrolling a uniform algorithm

Suppose X is the input set for a problem P. Assume that we have a notion of **input size**, which is a function

$$size : X \to \mathbb{N}. \tag{2}$$

where $size(x)$ is known as the **size** of $x \in X$. *We assume there are inputs of arbitrarily large size.* E.g., for the sorting program, X is the set of all sequences of real numbers; if $x \in X$ is a sequence of n elements, then $size(x) = n$. Then we have $X = \cup_{n\in\mathbb{N}} X_n$ where $X_n := \{x \in X : size(x) = n\}$. Note that X_n might be empty for arbitrarily large values of n. For instance, if our inputs are square matrices, and we measure size of a matrix by the number of entries, it follows that X_n is empty unless n is a square (i.e., $n = m^2$ for some m).

A **uniform algorithm** for P is one that accepts all $x \in X$. But algorithm A_n that accepts only inputs from X_n (for some $n \in \mathbb{N}$) is called a **finite program**. Putting together an infinite set of such finite programs,

$$A = \{A_n : n \in \mathbb{N}\}. \tag{3}$$

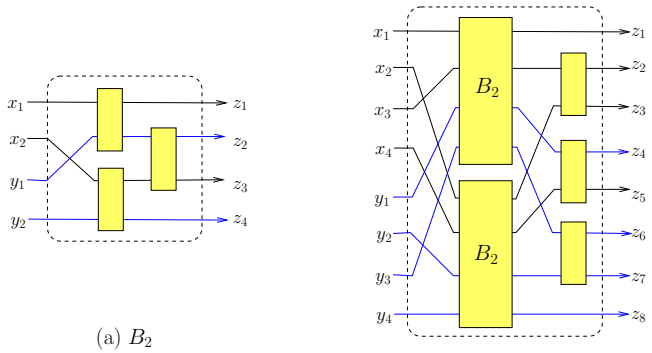
If each A_n solves the problem P for inputs of size n, then we call A a **non-uniform algorithm** for P. Intuitively, A is "non-uniform" because, *a priori*, there need not be any systematic method of generating the different A_n's.

For our merging problem, the input set X is now the set of all pairs (x, y) where x, y are sorted sequences. We define $size : X \to \mathbb{N}^2$ where $size(x, y) = (m, n)$ if x has length m and y length n. Clearly, the Merge Algorithm in ¶9 is a uniform algorithm for merging. The RAM model is called a "uniform model" because it permits the construction of uniform algorithms such as the Merge Algorithm. Pointer machines (see Chapter 6) and Turing machines are other examples of uniform models. In contrast, each program in the comparison tree model admits[3] a fixed size input. Thus the comparison tree model can only provide non-uniform algorithms such as (1). The relationship between complexity in uniform models and in non-uniform models is studied in complexity theory.

¶11. Merging in the Circuit Model. We now provide a comparator circuit B_n (after Batcher) to merge two sorted lists $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ of size n each. For simplicity, assume n is a power of 2, and the construction is recursive. For the base case, B_1 is just a comparator. Figure 3 shows B_2 and B_4.

In general, B_n is uses two copies of $B_{n/2}$ followed by a "layer" of $n-1$ comparators. The first (second) copy of $B_{n/2}$ takes as input the wires x_i and y_i where i is odd (even). For instance, the

[3] For this purpose, we say that the "input variables" for a comparison tree is the set $\{v_1, \ldots, v_n\}$ of variables that appear in some comparison in the tree. A sequence $(x_1, \ldots, x_n)$ of numbers is regarded as "input instance" under the assignments $v_i \leftarrow x_i$ for each $i = 1, \ldots, n$.



(a) B_2 (b) B_4

Figure 3: Merging Circuit

first copy of $B_{n/2}$ takes the inputs $(x_1, x_3, \ldots, x_{n-1})$ and $(y_1, y_3, \ldots, y_{n-1})$, while the second copy takes $(x_2, x_4, \ldots, x_n)$ and $(y_2, y_4, \ldots, y_n)$. Let their sorted outputs are $(u_1, \ldots, u_n)$ and $(v_1, \ldots, v_n)$ respectively. Next, let us do the following $n-1$ "compare-exchanges":

$$v_1 : u_2,\ v_2 : u_3,\ v_3 : u_4, \ldots, v_{n-1} : u_n.$$

After comparing $v_i : u_{i+1}$, we exchange (if necessary) the values of v_i and u_{i+1} so that the relation $v_i \leq u_{i+1}$ is true. Note that u_1 and v_n are not involved in these compare-exchanges. The final sorted output is

$$(u_1, v_1, u_2, v_2, u_3, \ldots, v_{n-1}, u_n, v_n).$$

The correctness of this circuit can be shown by the so-called "zero-one principle" (Exercise).

¶* 12. Merging in Tape Model. Let us now illustrate how one can design algorithms in the tape model. We assume some conventional programming language (or RAM model), but augmented with four **tape primitives**:

$$\texttt{EOT}(T), \quad \texttt{READ}(T, x), \quad \texttt{WRITE}(T, x), \quad \texttt{RESET}(T), \quad \texttt{ERASE}(T) \tag{4}$$

where T is a tape and x is a variable storing an item. The first primitive is a predicate $\texttt{EOT}(T)$ that returns `true` iff the head is at the end-of-tape. The remaining primitives are operations: $\texttt{READ}(T, x)$ copies the item under the current position on tape T into variable x, and advances the head to the next item. If the head is already at the end-of-tape, this operation is a "NO-OP" (nothing happens). The operation $\texttt{WRITE}(T, x)$ will write the value of x into the current head position (which may or may not be at the end of tape), and the head is advanced. Again, this is a NO-OP if the head is already at the physical end of the tape. $\texttt{RESET}(T)$ will rewind the tape to the very beginning of the tape. Thus the head is positioned at the first item, assuming the tape is non-empty. Finally, $\texttt{ERASE}(T)$ simply erases the contents of the tape from the current head position to the end of the tape. There is no head movement, and the tape contents, from the beginning of tape until the position before the current head position, are unchanged. Note that $\texttt{ERASE}(T)$ can be implemented easily by writing a special "end-marker" on the tape. Here are two examples of how to use these primitives.

E.g., to completely erase the contents of tape T, you do $\texttt{RESET}(T)$ followed by $\texttt{ERASE}(T)$.

E.g, to check if the tape T is empty, we first do $\texttt{RESET}(T)$ and then check if $\texttt{EOT}(T)$ is true.

We now provide a "tape algorithm" to do merging: assume that tapes T_1, T_2 contains two lists of sorted items (in non-decreasing order), and we want to merge the result into tape T_0. Here is the tape model implementation of the Merge Algorithm in ¶9. We use the variables x_i $(i = 1, 2)$ to store an item read from tape T_i. There are two Boolean variables, b_1 and b_2, where $b_i =$ `true` iff variable x_i holds an item that has not been output.

Tape Merge Algorithm:

▷ *Initialization: set-up x_i, b_i $(i = 1, 2)$*
  `RESET`(T_0), `ERASE`(T_0).
  `RESET`(T_1), `RESET`(T_2).
  $b_1 \leftarrow b_2 \leftarrow$ `true`.
  If `EOT`(T_1) then $b_1 \leftarrow$ `false`
  else `READ`(T_1, x_1).
  If `EOT`(T_2) then $b_2 \leftarrow$ `false`
  else `READ`(T_2, x_2).
▷ *Main Loop: both b_1 and b_2 are* `true`
  While $(b_1 \wedge b_2)$
    If $(x_1 \leq x_2)$
      `WRITE`(T_0, x_1).
      If `EOT`(T_1) then $b_1 \leftarrow$ `false`
      else `READ`(T_1, x_1).
    else
      `WRITE`(T_0, x_2).
      If `EOT`(T_2) then $b_2 \leftarrow$ `false`
      else `READ`(T_2, x_2).
▷ *Clean-Up: either b_1 or b_2 is* `false`.
  While (b_1)
    `WRITE`(T_0, x_1).
    If `EOT`(T_1) then $b_1 \leftarrow$ `false`
    else `READ`(T_1, x_1).
  While (b_2)
    $\cdots$ ◁ *Repeat the previous while-loop for T_2*

Note that this algorithm uses three tapes but in general, we can use any finite number. Our program can have any finite number of variables to store items: in this example, we use just two variables (x_1, x_2). In an Exercise, we ask you to design a tape algorithm for sorting. The goal is to minimize the number of passes (i.e., number of `RESET`'s).

¶13. Program Correctness. Recall our distinction between a "program" and an "algorithm". By definition, an algorithm is a program that is *correct* for a given problem. There is an area of computer science that formally studies program correctness, from the logical analysis of correctness concepts, to the introduction of tools to prove correctness. Correctness is also central for us, but we are less formal in our approach. It is usual to divide correctness into two parts: **partial correctness** and **halting**. The partial correctness part says that the algorithm gives the correct output *provided it halts*. The halting part simply asserts that the program always halts. Halting might sometimes be trivial (e.g., in our Merge algorithm above) but it can sometimes be highly nontrivial. We should say that there are some programs in which the "halting part" requires non-halting! For instance, if the program is an operating system, we

want to ensure that it never halts. Our definition of "computational problem" precludes such non-halting algorithms.

Exercises

Exercise 3.1: We interpreted the programs in Figure 2(a) and (b) as "algorithms for finding the maximum of $\{x, y, z\}$". But the notion of an "algorithm" is a semantical concept. So the same programs can be given different interpretations. Please give a different interpretation to these two programs. I.e., view them as solving a different problem.

NOTES: Regard the output at each leaf of a comparison tree as part of the "interpretation". So you may change the output at each leaf, but do not change the programs. However, x, y, z are still numbers – we are not interested in re-interpreting these numbers as time, strings, apples, etc. ♢

Exercise 3.2: (a) Extend the comparison tree in Figure 2(a) so that it sorts three input elements $\{x, y, z\}$.
(b) Extend the comparator circuit in Figure 2(a) so that it sorts three input elements $\{x, y, z\}$. ♢

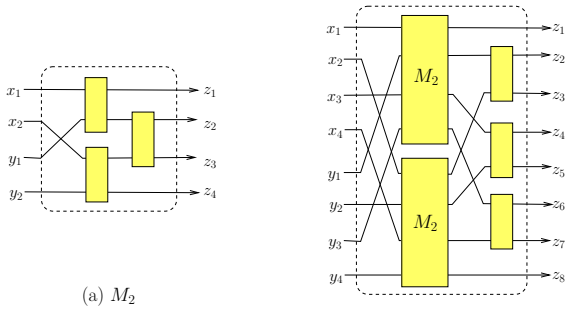
Exercise 3.3: Design tree programs for four elements a, b, c, d:
(a) To find the second largest element. The height of your tree should be 4 (the optimum).
(b) To sort the four elements. The height of your tree should be 5 (the optimum).
♢

Exercise 3.4: (a) Show that the median of 4 elements can be computed with 4 comparisons in the worst case. (b) Show that the median of 5 elements can be computed with 6 comparisons in the worst case. HINT: you could use your solution in part(a) for this part.
NOTE: the median of a set X of elements is the element of rank $\lceil |X|/2 \rceil$. An element has rank k if it is smaller than or equal to $k-1$ other elements and larger than or equal to $|X| - k$ other elements. Thus rank 1 is the largest, and rank $|X|$ is the smallest element. In case the elements are non-distinct, an element could have several ranks. For instance, if all the elements are identical, then each element could have ranks 1 to $|S|$. ♢

Exercise 3.5: Prove or disprove: the circuits M_2 and M_4 in Figure 4 correctly merges their inputs. ♢

Exercise 3.6: Assume n is a power of 2.
(a) Recursively describe a circuit S_n for sorting n numbers. You may use merging circuits M_k for any $k \leq n/2$ as subcircuits. Also, please draw the comparator circuit S_8. (b) Define the **delay** of a circuit to be the maximum number of comparators in a path from input node to output node. Suppose the maximum delay of the circuit M_n in part(a) is $\lg^2 n$. What is the delay of S_n in your construction (a)? ♢

Exercise 3.7: Prove the correctness of the circuits B_n in the text for merging two sorted lists of n numbers each. Use the "zero-one principle": *if a circuit correctly merges any* $0-1$ *inputs, then the circuit is correct.* ♢



(a) M_2 (b) M_4

Figure 4: Circuits M_2 and M_4

Exercise 3.8: Design a tree program to merge two sorted lists (x, y, z) and (a, b, c, d). The height of your tree should be 6. Can you prove that 6 is optimum? ♢

Exercise 3.9: ("Unrolling" a uniform algorithm into a comparison tree)
(a) Draw the comparison tree $T_{2,4}$ obtained by unrolling our Merge Algorithm on input (x_1, x_2) and (y_1, y_2, y_3, y_4).
(b) Draw the Hasse diagram associated with each node of the tree in Part (a). ♢

Exercise 3.10: Design a Tape Algorithm to sort. Assume that the input tape is T_0 containing a sequence of n items. Finally, you must output the sorted items in tape T_0. Besides T_0, it is sufficient to have two other tapes T_1 and T_2, but any additional number of tapes you like. Our goal is to minimize the number of `RESET`'s ($O(\log n)$ suffices).

HINT: Use some form of Merge Sort. Use pseudo-code as we have done in the text to describe the Tape Algorithm for Merging. One key concept is the notion of a **run** which is any longest contiguous sequence of items in a tape that is non-decreasing. E.g., $(1, 8, 2, 5, 9, 4)$ has three runs: $(1, 8), (2, 5, 9), (4)$. You want to merge the runs. But first you need to "distribute" the runs in the input tape into two other tapes, and then merge them: call this procedure $Distribute(T_0, T_1, T_2)$. ♢

Exercise 3.11: In the text, we described a procedure called $TapeMerge(T_1, T_2, T_0)$ to merge items in T_1 and in T_2 into T_0, *assuming* that the items in T_1, T_2 are already sorted. Let us explore what happens in case T_1 and T_2 have *many* runs. In the previous exercise, you also designed a procedure $Distribute(T_0, T_1, T_2)$ to distribute the runs in T_0 into T_1 and T_2 (alternately). For this question, assume that $Distribute(T_0, T_1, T_2)$ returns the number $k \geq 1$ of runs that was originally in T_0. Consider the following algorithm:

```
Input: Items in T_0. Initially T_1, T_2 empty.
  k ← Distribute(T_0, T_1, T_2).
  While (k > 1)
      TapeMerge(T_1, T_2, T_0)
      k ← Distribute(T_0, T_1, T_2)
```

Clearly, using $TapeMerge$ in this way is an abuse of our original intention. *But if this algorithm halts*, it would have successfully sorted the items in the original tape T_0. Our goal is to prove that it will, in fact, halt. REMARK: this algorithm was accidentally discovered by students when I asked them to design a tape sorting algorithm from scratch.

HINT: Consider the runs that appear in T_0 after $TapeMerge$. When does a new run appear in T_0? Can you bound the number of such events? ♢

Exercise 3.12: How do you speed up your algorithm in the previous exercise if you have 4 tapes? Note that "reducing speed" here means using fewer `RESET`'s in your algorithm. What if you have $k > 4$ tapes? ♢

Exercise 3.13: In the tape model, it is non-trivial to reverse the contents of a tape. For instance, if the input tape contains (a, b, c, d), we want the output tape to contain (d, c, b, a). Give an $O(\log n)$ pass algorithm to reverse a list of n items in a tape. HINT: the method is very similar to the tape merge or tape sort algorithm. ♢

End Exercises

§4. Complexity Model: How to assess algorithms?

We now have a suitable computational model for solving our problem. What is the criteria to choose among different algorithms within a model? For this, we need to introduce a **complexity model**.

In most computational models, there are usually natural notions of **time** and **space**. These are two examples of **computational resources**. Naturally, resources are scarce and algorithms consume resources when they run. We want to choose algorithms that minimize the use of resources. For this purpose, we shall focus on an algorithm's usage of only one resource, ignoring its behavior on the other resources. This resource is usually the time (occasionally space) resource. Thus we avoid studying the simultaneous usage of two or more resources, as this involves the more delicate issues of trade-offs between resources.

Next, for each primitive operation executing on a particular data, we need to know how much of the time resource is consumed. For instance, in `Java`, we could define each execution of the addition operation on two numbers a, b to use time $\log(|a| + |b|)$. Or again, the comparison $a : b$ of two integers in the comparison tree model may be charged $\log(|a| + |b|)$. But it would be simpler to say that these operation takes unit time, independent of a, b. This simpler version is our choice throughout these lectures: *each primitive operation takes unit time, independent of the actual arguments to the operation.*

After assigning a (time) cost to each primitive operation, for each algorithm A, and for each input instance x, we could now assign a number $T_A(x)$ which is the (time) complexity of algorithm A on input x. If X is the input domain for A, this defines the corresponding complexity function

$$T_A : X \to \mathbb{R}. \tag{5}$$

If B is another algorithm on domain X, we also have $T_B : X \to \mathbb{R}$. Thus allows us to compare A and B by comparing T_A and T_B. E.g., A is at least as good as B if for all $x \in X$, $T_A(x) \leq T_B(x)$. But to develop a complexity theory, we want a way to discuss the complexity of algorithm A without reference to other algorithms.

Usually, the input domain X has infinitely many elements, and there is a notion of "input size", as given by the function (2). Now we can measure resource usage as a function of input

size, using the following procedure. We define the **worst case complexity** of A to be the function $T_A : \mathbb{N} \to \mathbb{R}$ where

$$T_A(n) := \sup\{T_A(x) : x \in X, size(x) = n\}$$

Using sup illustrates one way to "aggregate" the set $\{T_A(x) : x \in X, size(x) = n\}$ of numbers. Instead of sup, we can also take the average to get **average complexity** In general, we may apply any aggregating function G and define

$$T_A(n) = G(\{T_A(x) : x \in X, size(x) = n\}).$$

In non-uniform models such as tree programs or comparator circuits, we may take A to a non-uniform program as in (3).

In summary, a **complexity model** is a specification of
(a) The computational resource (e.g., time, space),
(b) The cost (in terms of the computational resource) of primitive operations (e.g., unit cost, logarithmic cost),
(c) The input size function, $size : X \to \mathbb{N}$, and
(d) The method G of aggregating (e.g., worst case, average).
Once the complexity model is fixed, we can associate to each algorithm A a **complexity function**

$$T_A : \mathbb{N} \to \mathbb{R}. \tag{6}$$

We cannot overstate the theoretical advantage of the function (6) over (5). Complexity theory is founded[4] on functions such as (6). Moreover, (6) is possible thanks to the size function (2).

¶14. From Complexity of Algorithms to Complexity of Problems. The complexity function T_A concerns a single algorithm A. But properly speaking, *Complexity Theory is the study of the complexity of problems, not of algorithms per se.* If P is a problem, we need to consider the set of *all* T_A where A ranges over all algorithms A for P. Naturally, the A's must be programs in a fixed computational model. Among all these algorithms, it would be nice if there exists an algorithm A^* for P whose complexity T_{A^*} is "optimal". In general, optimal algorithms may not exist. But for non-uniform algorithms, we can always define the optimal algorithm $A^* = \{A^*_n : n \in \mathbb{N}\}$ since each A^*_n can be chosen to have minimal height. The complexity of such an optimal algorithm may be called the **inherent complexity** of the problem P (relative to the computational model). We next introduce two such examples.

¶15. Example (T1). Inherent Complexity of Sorting. Consider the comparison tree model for sorting. For each $n \in \mathbb{N}$, let $S(n) := \min_A T_A$ where A ranges over all comparison trees that sort n elements (recall that T_A is the height of the tree A). The function $S : \mathbb{N} \to \mathbb{N}$ so defined is called the **inherent complexity of sorting**. For instance, it is easy to see that $S(1) = 0$ and $S(2) = 1$. It is "inherent" because it is not a function of a single algorithm, but speaks to all possible algorithms for sorting. This is because even uniform algorithms can be unrolled into a non-uniform comparison tree algorithm.

We now prove our first non-trivial result in this chapter. Start with the simple observation: *any tree program A to sort n elements must have at least $n!$ leaves.* This is because A must have at least one leaf for each possible sorting outcome, and there are $n!$ outcomes when the input elements are all distinct. But a binary tree A of height h has at most 2^h leaves. Hence $2^h \geq n!$ or $h \geq \lg(n!)$. This proves:

[4] In situations where there is no suitable size functions, e.g., for $X = \mathbb{R}$, only an impoverished complexity theory can be developed.

Lemma 1 (Information-Theoretic Bound) *Every tree program for sorting n elements has height at least* $\lg(n!)$*, i.e.,*

$$S(n) \geq \lg(n!). \tag{7}$$

This result is called the **Information Theoretic Bound** (ITB) for sorting. To use (7) effectively, it helpful to know $\lg(n!)$ is roughly $n \lg n$ (see Exercises for more precise formulations). Thus, it tells use that you need at least "$n \lg n$" comparisons to sort n elements.

Since $S(n)$ is an integer, the ITB bound (7) is equivalently to $S(n) \geq \lceil \lg(n!) \rceil$. Defining $\texttt{ITB}(n) := \lceil \lg(n!) \rceil$, here are its first 10 values:

n:	1	2	3	4	5	6	7	8	9	10
$\texttt{ITB}(n)$:	0	1	3	5	7	10	13	16	19	22

E.g., you need at least 22 comparisons to sort 10 elements. This deceptively simple result is really quite deep: to appreciate this, try proving by direct arguments that, in the worst case, you need more than four comparisons to sort four elements. The wise words of Fraleigh[5] (see margin) extend to inequalities like the ITB. How good is the ITB lower bound on $S(n)$? Trivially, we can see that $S(1) = 0$ and $S(2) = 1$. What about $n = 3$? By ITB, $S(3) \geq \lg 3!$, so $S(3) \geq \lceil \lg 6 \rceil = 3$. Let us check the next simplest case, $S(3)$. It is easy to see that you can sort three elements in at most 3 comparisons: if you are given distinct x, y, z then you can begin by comparing $x : y$ and $x : z$. If you are lucky, this might end up sorting the elements (either $y > x > z$ or $z > x > y$). Otherwise one more comparison $y : z$ will sort the input. *This is obvious if you use Hasse diagrams!* This proves $S(3) \leq 3$. Combined with the ITB, we conclude that $S(3) = 3$, and so the ITB bound is tight.

"Never underestimate a theorem that counts something" – J.B. Fraleigh

Note how the proof of $S(3) = 3$ requires two distinct arguments: an upper bound argument $S(3) \leq 3$ amounts to providing an algorithm. The lower bound argument $S(3) \geq 3$ comes from ITB. In a small way, this is what complexity theory is all about – getting good upper (by studying algorithms) and lower bounds (by devising impossibility arguments) on computational problems. It is known that the ITB bound is optimal for $n \leq 31$.

Complexity Theory in a nutshell!

Open problem: determine $S(32)$

¶16. Example (T2). Inherent Complexity of Merging. We similarly define the **inherent complexity of merging** to be the function $M : \mathbb{N}^2 \to \mathbb{N}$ where $M(m, n)$ is the minimum height of any comparison tree for merging two sorted lists of sizes m and n, respectively. Let us prove the following upper and lower bounds:

Lemma 2 (Inherent Bounds on Merging)

$$(a) \quad M(m,n) \leq m + n - 1, \tag{8}$$

$$(b) \quad M(m,n) \geq 2\min\{m,n\} - \delta(m = n), \tag{9}$$

$$(c) \quad M(m,n) \geq \lg \binom{m+n}{m}. \tag{10}$$

where $\delta(m = n) = 1$ *if* $m = n$ *and* $\delta(m = n) = 0$ *otherwise.*

[5] Fraleigh was referring to Lagrange's theorem on finite groups in **A First Course in Abstract Algebra**, Addison-Wesley 1969, p. 93, and other similar counting theorems in algebra. Since learning these words as an undergraduate, its wisdom has only grown on me over time.

Proof. The upper bound (8) comes from the Merge Algorithm in ¶9. The idea of the proof is to associate one comparison for each one output in the main loop. More formally, we devise a simple **charging scheme** whereby each comparison that the algorithm makes is "charged" to the element that is output as a result of the comparison. But you cannot charge more than the number of output elements. This gives an upper bound of $\leq m+n$ comparisons. We improve this bound by observing that the last element can be output without any comparison. Hence we obtain the sharper upper bound of $m+n-1$. This charging argument is a very elementary example of what we call an **amortized analysis** in Chapter 6.

The lower bound (9) comes from the following input instance: assume the input is $x_1 < x_2 < \cdots < x_m$ and $y_1 < \cdots < y_n$ where $m \geq n$ and

$$x_1 < y_1 < x_2 < y_2 < x_3 < \cdots < x_n < y_n (< x_{n+1} < \cdots < x_m).$$

Let us rename the first $2n$ elements as

$$z_1 < z_2 < z_3 < z_4 < z_5 < \cdots < z_{2n-1} < z_{2n}$$

where $z_{2i-1} = x_i$ and $z_{2i} = y_i$ $(i = 1, \ldots, n)$. We claim that the comparison $z_i : z_{i+1}$ must be made for each $i = 1, \ldots, 2n-1$.

> The claim follows from a simple fact about partial orders (see Appendix for definition). A relationship $x < y$ in a partial order P is **essential** if it cannot be deduced from other relationships in P. In the comparison model, every essential relationship must be determined by a comparison. In particular, the relationships $z_i < z_{i+1}$ are essential.
> These primitive relationships constitute the edges of a directed graph called the **Hasse diagram** of P. In practice, it is very helpful to draw such diagrams to represent P for small examples.

This yields a lower bound of $2n-1$ comparisons. In case $m > n$, there is at least one more comparison to be made, between y_n and x_{n+1}. So if $m > n$, we need at least $2n$ comparisons. This proves $M(m,n) \geq 2n - \delta(m,n)$, where $n = \min\{m,n\}$. This method of proving lower bounds is simple form of what are called **adversary arguments** in Chapter XII, where you imagine a 2-player game between the algorithm and an adversary.

Finally, the lower bound (10) is the **information-theoretic bound** (ITB) for merging (analogous to (7) for sorting). In proof, there are $\binom{m+n}{n}$ ways of merging the two sorted lists. To see this, note that each sorted list of $m+n$ elements is uniquely determined once we know the m positions that are filled by elements that come from the list of size m. There are $\binom{m+n}{m}$ ways of choosing these positions. This concludes our proof of Lemma 2. **Q.E.D.**

As corollary of the upper and lower bounds, we obtain some exact bounds for the complexity of merging:

$$M(m,m) = 2m-1$$

and

$$M(m,m+1) = 2m.$$

Thus the uniform algorithm is optimal in these cases. More generally, $M(m,m+k) = 2m+k-1$ for $k = 0, \ldots, 4$ and $m \geq 6$ (see [3] and Exercise). These bounds are for inputs where $|m-n|$ is

a small constant. Now consider the other extreme situation where $|m-n|$ is as large as possible: $M(1,n)$. In this case, the information theoretic bound says that $M(1,n) \geq \lceil \lg(n+1) \rceil$ (why?). Also, by binary search, this lower bound is tight (Exercise). Hence we now know another exact value:

$$M(1,n) = \lceil \lg(n+1) \rceil . \tag{11}$$

A non-trivial result from Hwang and Lin says

$$M(2,n) = \lceil \lg 7(n+1)/12 \rceil + \lceil \lg 14(n+1)/17 \rceil .$$

Thus we have two distinct methods for proving lower bounds on $M(m,n)$: the adversary method is better when $|m-n|$ is small, and the information theoretic bound is better when this gap is large. The exact value of $M(m,n)$ is known for several other cases, but a complete description of this complexity function remains an open problem.

Student: *I thought lower bounds on $S(n)$ is about the "best case" complexity of sorting*

¶* 17. Best Case Complexity. Although our main interest is in worst-case complexity, it is useful to briefly consider the notion of "best case complexity". Note that contrary to what some students think, lower bounds on $S(n)$ is *still* about worst case complexity, not about best case complexity.

Again, if A_n is a tree program that accepts inputs of size n, we define the **best case complexity** of A_n to be the length of the *shortest path* from the root of A_n to a leaf. We can apply this concept to sorting and to merging. Define $S'(n)$ to be the best case complexity for sorting n elements: $S'(n) := \min\{T'_{A_n}\}$ where A_n range over all tree programs that sort n elements. Similarly, define $M'(m,n)$ to be the best case complexity for merging m elements with n elements. We claim:

Student: *I thought $M'(m,n) = \min\{m,n\} - \delta(m,n)$*

$$S'(n) = n-1, \qquad M'(m,n) = 1. \tag{12}$$

To see $S'(n) = n-1$, we see that if the output of sorting is $(z_1, \ldots, z_n)$, then we must have made the comparisons $z_i : z_{i+1}$ for $i = 1, \ldots, n-1$. To see that $M'(m,n) = 1$, note that on input $(x_1, \ldots, x_m)$ and $(y_1, \ldots, y_n)$, we may be able to get away with a single comparison $x_m : y_1$.

¶* 18. Complexity of Comparator Circuits. Let C be a comparator circuit. There are two complexity measures of interest, **delay** $D(C)$ and **size** $S(C)$: $D(C)$ is the length of the longest path from an input node to an output node, and $S(C)$ is the number of comparators in C. Since values move in parallel, $D(C)$ can be thought off as "parallel time". In 2968, Batcher introduced circuits (see Exercise) for sorting n numbers with delay $\mathcal{O}(\log^2 n)$ and size $\mathcal{O}(n \log^2 n)$. In 1983, Ajtai, Komlós and Szemerédi made a major breakthrough by proving the ex when sorting circuits with delay $\mathcal{O}(\log n)$ and size $\mathcal{O}(n \log n)$. It is an understatement to say that this result is impractical. Knuth noted that unless n exceeds the number of atoms in the universe, the size of Batcher's circuit is smaller; see the blog of Lipton In 2004, Goodrich made another breakthrough by providing a sorting circuit of size $O(n \log n)$ that seems practical. We return to this topic in Chapter 12.

¶* 19. Other Complexity Measures. We briefly look at some other kinds of complexity measures.

- In computational geometry, it is often useful to take the output size into account. The complexity function would now take at least two arguments, $T(n, k)$ where n is the input size, but k is the output size. This is the **output-sensitive complexity model**.

- Another kind of complexity measure is the **size** of a program. In the RAM model, this can be the number of primitive instructions. We can measure the complexity of a problem P in terms of the size $s(P)$ of the smallest program that solves P. This complexity measure assigns a single number $s(P)$, not a complexity function, to P. This **program size measure** is an instance of **static complexity measure**; in contrast, time and space are examples of **dynamic complexity measures**. Here "dynamic" ("static") refers to fact that the measure depends (does not depend) on the running of a program. Complexity theory is mostly developed for dynamic complexity measures.

- The comparison tree complexity model ignores all the other computational costs except comparisons. In most situations this is well-justified. But it is possible[6] to conjure up ridiculous algorithms which minimize the comparison cost, at an exorbitant cost in other operations.

- The size measure is relative to representation. Perhaps the key property of size measures is that *there are only finitely many objects up to any given size*. Without this, we cannot develop any complexity theory. If the input set are real numbers, $\mathbb{R}$, then it is very hard to give a suitable size function with this property. This is the puzzle of real computation.

Exercises

Exercise 4.1: How many comparisons are required in the worst case to sort 10 elements in the comparison tree model? In other words, give a lower bound on $S(10)$. HINT: to do this computation by hand, it is handy to know that $10! = 3,628,800$ and $2^{20} = 1,048,576$. ♢

Exercise 4.2: Show that $\lg(n!) \geq (n/2)\lg(n/2)$. HINT: use the fact that $\lg(ab) = \lg(a) + \lg(b)$ to write $\lg(n!)$ as a summation. Then discard the smallest $n/2$ elements. Since $(n/2)$ might not be an integer, use the relation $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. ♢

Exercise 4.3: How good is the information theoretic lower bound? In other words, can we find upper bounds that match information-theoretic lower bound? We know it is tight for $S(3)$. What about $S(4)$? ♢

Exercise 4.4: (a) Give a lower bound for $S(5)$.
(b) Show that if the first two comparisons should involve only three input elements, then you need a total of at least 8 comparisons in the worst case.
(c) Using the insight from (b), give an optimal algorithm to sort 5 elements. ♢

Exercise 4.5: Give an upper and a lower bound for $S(1000)$, the complexity of sorting 1000 elements. NOTE: we are asking for two numbers. Justify how you obtain these two

[6] My colleague, Professor Robert Dewar gives the following example: given n numbers to be sorted, we first search for all potential comparison trees for sorting n elements. To make this search finite, we only evaluate comparison trees of height at most $n\lceil \lg n \rceil$. Among those trees that we have determined to be able to sort, we pick one of minimum height. Now we run this comparison tree on the given input.

numbers. Your numbers must be *explicit* (in decimal notation like 1234), not an expression like 1000^2 $\lceil \lg 1000 \rceil$. You may use computer programs or calculators, etc, but tell us how you do it. If you use computers, discuss how you can be confident despite rounding errors in the computation. Stirling's formula might be useful to know. ♢

Exercise 4.6: Give upper and lower bounds for $S(100)$. Similar to the previous question. ♢

Exercise 4.7: The following is a variant of the previous exercise. Is it always possible to sort n elements using a comparison tree with $n!$ leaves?
(a) Let $L(n)$ be the minimal number of leaves in a comparison tree that sorts n elements. Clearly, $L(n) \geq n!$. Show that $L(n) = n!$.
(b) Let us make the problem a bit harder: let $L^*(n)$ be the minimal number of leaves in a comparison tree that sorts n elements in (optimal) height $S(n)$. Clearly $L^*(n) \geq L(n) = n!$. Give the best upper bound you can for $L^*(n)$ when $n = 3, 4, 5$. ♢

Exercise 4.8: (a) Consider a variant of the unit time complexity model for the integer RAM model, called the **logarithmic time complexity model**. Each operand takes time that is logarithmic in the address of the register and logarithmic in the size of its operands. What is the relation between the logarithmic time and the unit time models?
(b) Is this model realistic in the presence of the arithmetic operators (ADD, SUB, MUL, DIV). Discuss. ♢

Exercise 4.9: Describe suitable complexity models for the "space" resource in integer RAM models. Give two versions, analogous to the unit time and logarithmic time versions. What about real RAM models? ♢

Exercise 4.10: Prove the claim (11) that $M(1, n) \leq \lceil \lg(n + 1) \rceil$. ♢

Exercise 4.11: Give your best upper and lower bounds for $M(2, 10)$. For upper bound, please give an explicit method. ♢

Exercise 4.12: Prove that $M(m, m + i) = 2m + i - 1$ for $i = 2, 3, 4$ for $m \geq 6$. ♢

Exercise 4.13: Prove that $M(k, m) \geq k \lg_2(m/k)$ for $k \leq m$. HINT: split the list of length m into three sublists of roughly equal sizes. ♢

Exercise 4.14: Open problem: determine $M(m, 3)$ and $M(m, m + 5)$ for all m. ♢

Exercise 4.15: Let C be a comparator circuit for a problem P. If C has h comparators, show that there exists is a comparison tree A with height h that solves problem P. You must argue in what sense P is solved. ♢

Exercise 4.16: Describe time and space complexity models for the comparator circuit model in ¶7. Then define $T(n)$ and $S(n)$ as the inherent time and inherent space to sort n

numbers in this model. HINT: "Time" is the maximum number of comparisons along any path, and "space" is the number of comparators. Derive bounds on $T(n)$ and $S(n)$. ♢

Exercise 4.17: Suppose $X_1, \ldots, X_n$ are n sorted lists, each with k elements. Show that the complexity of sorting the set $X = \bigcup_{i=1}^{n} X_i$ is $\Theta(nk \log n)$. ♢

End Exercises

§5. Algorithmic Techniques: How to design efficient algorithms

Now that we have some criteria to judge algorithms, we begin to design algorithms that are "efficient" according to such criteria. There emerges some general paradigms of algorithms design:
(i) Divide-and-conquer (e.g., merge sort)
(ii) Greedy method (e.g., Kruskal's algorithm for minimum spanning tree)
(iii) Dynamic programming (e.g., multiplying a sequence of matrices)
(iv) Incremental method (e.g., insertion sort)

Let us briefly outline the merge sort algorithm to illustrate the divide-and-conquer paradigm: Suppose you want to sort an array A of n elements. Here is the **Merge Sort** (or Mergesort) algorithm on input A:

Merge Sort Algorithm
Input: An array A with $n \geq 1$ numbers.
Output: The sorted array A containing these numbers, but in non-decreasing order.
0. (Basis) If $n = 1$, return the array A.
1. (Divide) Divide the elements of A into two subarrays B and C of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ each.
2. (Recurse) Recursively, call the Merge Sort algorithm on B. Do the same for C.
3. (Conquer) Merge the sorted arrays B and C into the array A

It is important to note that this is a recursive algorithm: the algorithm calls itself on smaller size inputs. E.g., to Merge Sort (a, b, c, d), you have to recursively Merge Sort (a, b) and (c, d). Besides recursion, there is only one non-trivial step in this algorithm, the Conquer Step which merges two sorted arrays. The subalgorithm for merging was already present in ¶9.

There are many variations or refinements of these paradigms. E.g., Kirkpatrick and Seidel [2] introduced a form of divide-and-conquer (called "marriage-before-dividing") that leads to an output-sensitive convex hull algorithm. There may be domain specific versions of these methods. E.g., plane sweep is an incremental method suitable for problems on points in Euclidean space.

Closely allied with the choice of algorithmic technique is the choice of *data structures*. A data structure is a representation of a complex mathematical structure (such as sets, graphs or

matrices), together with algorithms to support certain querying or updating operations. For instance, to implement recursive algorithms such as Merge Sort above, we will need the use of a "stack" to organize the recursive calls. A stack is an example of a basic data structure. The following are a list of such basic data structures.

(a) Linked lists: each list stores a sequence of objects together with operations for (i) accessing the first object, (ii) accessing the next object, (iii) inserting a new object after a given object, and (iv) deleting any object.

(b) LIFO, FIFO queues: each queue stores a set of objects under operations for insertion and deletion of objects. The queue discipline specifies which object is to be deleted. There are two[7] basic disciplines: last-in first-out (LIFO) or first-in first-out (FIFO). Note that recursion is intimately related to LIFO.

(c) Binary search trees: each tree stores a set of elements from a linear ordering together with the operations to determine the smallest element in the set larger than a given element. A dynamic binary search tree supports, in addition, the insertion and deletion of elements.

(d) Dictionaries: each dictionary stores a set of elements and supports the operations of (i) inserting a new element into the set, (ii) deleting an element, and (iii) testing if a given element is a member of the set.

(e) Priority queues: each queue stores a set of elements from a linear ordering together with the operations to (i) insert a new element, (ii) delete the minimum element, and (iii) return the minimum element (without removing it from the set).

Exercises

Exercise 5.1: (a) Design an incremental sorting algorithm based on the following principle: assuming that the first m elements have been sorted, try to add ("insert") the $m+1$st element into the first m elements to extend the inductive hypothesis. Moreover, assume that you do all these operations using only the space in the original input array.

(b) If the number n of elements to be sorted is small (say $n < C$), this approach can lead to a sorting algorithm that is faster than Merge Sort. Intuitively it is because Merge Sort uses recursion that has non-trivial overhead cost. So a practical implementation of Merge Sort might switch an incremental sorting method as in part(a) when $n < C$. Design such a hybrid algorithm that combines the Merge Sort algorithm with your solution in (a).

(c) Implement the Merge Sort Algorithm, your incremental sorting algorithm of part(a), and the hybrid algorithm in part(b). Try to see if you can experimentally verify our remarks in (b), and determine the constant C. ♢

End Exercises

§6. Analysis: How to estimate complexity

[7] A discipline of a different sort is called GIGO, or, garbage-in garbage-out. This is really a law of nature.

We have now a measure T_A of the complexity of our algorithm A, relative to some complexity model. Unfortunately, the function T_A is generally too complex to admit a simple description, or to be expressed in terms of familiar mathematical functions. Instead, we aim to give upper and lower bounds on T_A. This constitutes the subject of **algorithmic analysis** which is a major part of this book. The tools for this analysis depend to a large extent on the algorithmic paradigm or data structure used by A. We give two examples.

¶20. Example (D1) Divide-and-Conquer. If we use divide-and-conquer then it is likely we need to solve some recurrence equations. In our Merge Sort algorithm, assuming n is a power of 2, we obtain the following recurrence:

$$T(n) = 2T(n/2) + Cn$$

for $n \geq 2$ and $T(1) = 1$, and $C \geq 1$ is some constant determined by the complexity of merging. Here $T(n) = T_A(n)$ is the (worst case) number of comparisons needed by our algorithm A to sort n elements. The solution is $T(n) = \Theta(n \log n)$. In the next chapter, we study techniques to obtain such solutions.

¶21. Example (D2) Amortization. If we employ certain data-structures that might be described as "lazy" then amortization analysis might be needed. Let us illustrate this with the problem of maintaining a binary search tree under repeated insertion and deletion of elements. Ideally, we want the binary tree to have height $\mathcal{O}(\log n)$ if there are n elements in the tree. There are a number of known solutions for this problem (see Chapter 3). Such a solution achieves the optimal logarithmic complexity for *each* insertion/deletion operation. But it may be advantageous to be lazy about maintaining this logarithmic depth property: such laziness may be rewarded by a simpler coding or programming effort. The price for laziness is that our complexity may be linear for individual operations, but we may still hope to achieve logarithmic cost in an "amortized" sense (thought of as a kind of averaging). To illustrate this idea, suppose we allow the tree to grow to non-logarithmic depth as long as it does not cost us anything (*i.e.*, there are no queries on a leaf with big depth). But when we have to answer a query on a "deep leaf", we take this opportunity to restructure the tree so that the depth of this leaf is now reduced (say halved). Thus repeated queries to this leaf will make it shallow. The cost of a single query could be linear time, but we hope that over a long sequence of such queries, the cost is amortized (averaged) to something small (say logarithmic). This technique prevents an adversary from repeated querying of a "deep leaf". But how do we account for the first few queries into some "deep leaves" which have linear costs? To anticipate such expenses, the idea is to "pre-charge" those initial insertions that lead to this inordinate depth. Using a financial paradigm, we put the pre-paid charges into some bank account. Then the "deep queries" can be paid off by withdrawing from this account. Amortization is both an algorithmic paradigm as well as an analysis technique. This will be treated in Chapter 6.

§7. Asymptotics: How robust is the model?

This section contains important definitions for the rest of the book.

Take note!

Let us review what we have done so far: we started with a problem P (§2), selected an appropriate computational model (§3) and an associated complexity model (§4), and designed an algorithm A for P in our model (§5). As we next embark on the problem of analyzing the

complexity function T_A of our algorithm (§6) in order to understand how good or efficient is our algorithm. We quickly realize that T_A will generally be too complex to determine exactly. But looking back at this process, we are bound to find many arbitrary choices that affects T_A. For instance, would a simple change in the set of primitive operations drastically change the complexity of your solution? Or what if we charge two units of time for some of the operations? Of course, there is no end to such revisionist afterthoughts. What we are really seeking is a certain robustness or invariance in our results. This section addresses this important concern.

¶22. Partial and total functions. Let $f : D \to R$ be a function, where D is called the **domain** and R the **range**. In ordinary discourse, this is understood to mean that for every $x \in D$, the function f returns a value $f(x) \in R$. We are now going to consider a slightly more general concept. We call $f : D \to R$ a **partial function** if for all $x \in D$, either $f(x)$ is either **defined**, in which case $f(x)$ represents an element of R, or else $f(x)$ is **undefined**, and does not represent anything. We shall write write $f(x) =\uparrow$ if $f(x)$ is undefined, and write $f(x) =\downarrow$ if it is defined. The partial function f is said to be a **total function** if for all $x \in D$, $f(x)$ is defined (and hence $f(x) \in R$). In other words, total functions are the kind of functions we ordinarily assume. But in the presence[8] of partial functions, we need to give it a name.

$\uparrow$ may be seen as a special value, but $\downarrow$ is only a surrogate for all other values!

¶23. What is a complexity function? In this book, we call a partial function of the form

$$f : \mathbb{R}^n \to \mathbb{R}$$

a **complexity function**. Usually, we have $n = 1$. We use complexity functions to quantify the complexity of algorithms. Why do we consider *partial* functions for complexity functions? For one thing, many functions of interest are only defined on positive integers. For example, the running time $T_A(n)$ of an algorithm A that takes discrete inputs is a partial real function (normally defined only when n is a natural number). Of course, if the domain of T_A is taken to be $\mathbb{N}$, then $T_A(n)$ might perhaps be total. Still, we prefer to use $\mathbb{R}$ as the domain of $T_A(n)$. This is because we often use functions such $f(n) = n/2$ or $f(n) = \sqrt{n}$, to bound our complexity functions, and these are naturally defined on the real domain; all the tools of analysis and calculus become available to analyze such functions. Many common real functions such as $f(n) = 1/n$ or $f(n) = \log n$ are partial functions because $1/n$ is undefined at $n = 0$ and $\log n$ is undefined for $n \leq 0$.

We have to be careful about operations on partial functions, and when they are used to define predicates. We have a general rule for composition of two partial functions $f, g : \mathbb{R} \to \mathbb{R}$:

$$g(x) =\uparrow \text{ implies } f(g(x)) =\uparrow . \tag{13}$$

More simply: $f(\uparrow) =\uparrow$ is always true. In general, if any argument of a function is undefined, then the value of the function is undefined.

¶24. Partial Predicates. For any set D, a partial function $P : D \to \{0, 1\}$ is called a **partial predicate** over D. We say the predicate P **holds at** $x \in D$ if $P(x) = 1$. So 1 is the "true" value and 0 is the "false" value. The partial predicate P is **valid** if for all $x \in D$, either $P(x) =\uparrow$ or $P(x) = 1$. If $P(x) =\uparrow$ for all $x \in D$, then we say P is **vacuously** valid. Partial predicates arise naturally from relations among partial functions. If f, g are complexity functions, then the relation "$f \leq g$" represents the partial predicate $P : \mathbb{R} \to \{0, 1\}$ where

[8] We remark that the literature sometimes use the notation $f : D \dashrightarrow R$ to indicate that f is a partial function. However, we shall not use this "$\dashrightarrow$" notation.

$P(x) =\uparrow$ if $f(x) =\uparrow$ or $g(x) =\uparrow$; otherwise, $P(x) =\downarrow$. Naturally, when $P(x) =\downarrow$, we have $P(x) = 1$ iff $f(x) \leq g(x)$.

If $P_i : D \to \{0, 1\}$ are partial predicates ($i = 0, 1$), then so are $\neg P_i$, $P_0 \vee P_1$ and $P_0 \wedge P_1$ (recall our general rule (13) about composition of partial functions).

How to quantify over partial predicates

Quantification over partial predicates is defined as follows: The sentence "$(\forall x)P(x)$" holds iff for all $x \in D$, either $P(x) =\uparrow$ or $P(x) = 1$. Similarly "$(\exists x)P(x)$" holds iff there is some $x \in D$ such that $P(x) =\downarrow$ and $P(x) = 1$. E.g., if P is the always undefined predicate, then $(\exists x)P(x)$ is false. De Morgan's law for quantifiers say that

$$\left.\begin{array}{rcl} \neg(\forall x)P(x) & \equiv & (\exists x)\neg P(x). \\ \neg(\exists x)P(x) & \equiv & (\forall x)\neg P(x). \end{array}\right\} \tag{14}$$

It is not hard to see that (14) holds even when P is a partial predicate.

¶25. Designated variable and Anonymous functions. In general, we will write "n^2" and "$\log x$" to refer to the functions $f(n) = n^2$ or $g(x) = \log x$, respectively. Thus, the functions denoted n^2 or $\log x$ are **anonymous** (or, perhaps more accurately, "self-naming"). This convention is very convenient, but it relies on an understanding that "n" in n^2 or "x" in $\log x$ is the **designated variable** in the expression. For instance, the anonymous complexity function $2^x n$ is a linear function if n is the designated variable, but an exponential function if x is the designated variable. *The designated variable in complexity functions, by definition, range over real numbers.* This may be a bit confusing when the designated variable is "n" since in mathematical literature, n is usually a natural number.

n might be a real variable!

¶26. Robustness or Invariance issue. Let us return to the robustness issue which motivated this section. The motivation was to state complexity results that have general validity, or independent of many apparently arbitrary choices in the process of deriving our results. There are many ways to achieve this: for instance, we can specify complexity functions up to "polynomial smearing". More precisely, two real functions f, g are said to be **polynomially equivalent** if for some $c > 0$, $f(n) \leq cg(n)^c$ and $g(n) \leq cf(n)^c$ for all n large enough. Thus, $\sqrt{n}$ and n^3 are polynomially equivalent according to this definition. This is *extremely* robust but alas, too coarse for most purposes. The most widely accepted procedure is to take two smaller steps:

two steps towards invariance

- Step 1: We are interested in the eventual behavior of functions. E.g., if $T(n) = 2^n$ for $n \leq 1000$ and $T(n) = n$ for $n > 1000$, then we want to regard $T(n)$ as a linear function.
- Step 2: We distinguish functions only up to multiplicative constants. E.g., $n/2$, n and $10n$ are indistinguishable,

These two decisions give us most of the robustness properties we desire, and are captured in the following language of asymptotics.

Where is Asymptopia? a far, far away land, where everything is BIG.

¶27. Eventuality. This is Step 1 in our search for invariance. Let $P : \mathbb{R} \to \{0, 1\}$ be a (partial) real predicate. We say P holds **eventually**, denoted "P (ev.)", if $P(x)$ holds for all x large enough. More precisely:

$$(\exists x_0)(\forall x)[x \geq x_0 \Rightarrow P(x)]. \tag{15}$$

Instead of "P (ev.)", we may also write

$$P(x) \text{ (ev. } x).$$

to explicitly show the role of the variable x. According to our rules for quantifying over partial predicates, "$(\forall x)$" in (15) really says "$(\forall x$ such that $P(x) =\downarrow)$".

A typical example is when $P(x)$ is the predicate "$f(x) \leq g(x)$" defined by two complexity functions f, g. Then we say "$f \leq g$ (ev.)" if $f(x) \leq g(x)$ holds for all x large enough. More precisely,

$$(\exists x_0)(\forall x)[x \geq x_0 \Rightarrow f(x) \leq g(x)].$$

The "$(\forall x)$" in this statement really says "$(\forall x$ such that $f(x) =\downarrow$ and $g(x) =\downarrow)$".

By not caring about the behavior of complexity function over some initial values, our complexity bounds becomes robust against the following **table-lookup trick**. If A is any algorithm, relative to to any given finite set S of inputs, we can modify A so that if $x \in S$, then the answer for x is obtained by a table lookup; otherwise, the answer is computed by running A on x. The modified algorithm A' might be much faster than A for all $x \in S$, but it will have the same "eventual" complexity as A. Thus, the complexity of A and A' are indistinguishable using our eventuality criterion.

¶28. Infinitely Often. The concept of eventuality is intimately connected with the concept of **infinitely often** ("i.o." for short). Given a real predicate $P(x)$, we say P holds **infinitely often**, written

$$P(x) \text{ (i.o. } x) \tag{16}$$

(or simply "P (i.o.)" when x is understood) if

$$(\forall x_0)(\exists x > x_0)[P(x)].$$

We have this logical equivalence:

$$\neg(P(x) \text{ (ev. } x)) \equiv (\neg P(x)) \text{ (i.o. } x) \tag{17}$$

In other words, there is a De Morgan-like law governing the pair of conditions (ev. x) and (i.o. x).

For instance, for complexity functions f and g, we say "$f \leq g$ (i.o.)" if for all x_0, there exists $x > x_0$ such that $f(x) =\downarrow$ and $g(x) =\downarrow$ and $f(x) \leq g(x)$. Note[9] that an "infinitely often" (i.o.) statement is equivalent to the negation of an "eventually" statement:

$$\neg[P(x) \text{ (ev. } x)] \equiv [\neg P(x)] \text{ (i.o. } x) \tag{18}$$

Most natural functions f in complexity satisfy some rather natural properties:

- f is eventually defined, $f(x) =\downarrow$ (ev.).
- f is eventually non-negative, $f \geq 0$ (ev.).

When these properties fail, our intuitions about complexity functions may go wrong.

[9] A possibly confusion is this: if P (i.o.), then it is true that there are infinitely many values of x's such that $P(x)$ holds. However, this is not sufficient. We need the x's to increase without bound.

¶29. Domination. We now take Step 2 towards invariance. We say f **dominates** g, written

$$f \succeq g,$$

if there exists $C > 0$ such that $C \cdot f \geq g$ (ev.). The symbol '$\succeq$' is intended to evoke the '$\geq$' connection. In particular, it should suggest the transitivity property: $f \succeq g$ and $g \succeq h$ implies $f \succeq h$. Of course, the reflexivity property holds: $f \succeq f$. We can also write "$g \preceq f$" instead of $f \succeq g$. If $f \succeq g$ and $g \succeq f$ then we write

$$f \asymp g.$$

Clearly $\asymp$ is an equivalence relation. The equivalence class of f is (essentially) the Θ-**order** of f; more on this below. If $f \succeq g$ but not $g \succeq f$ then we write

$$f \succ g$$

ans say that f **strictly dominates** g. E.g., $n^2 \succ n \succ 1 + \frac{1}{n}$. An interesting example is $\sin x \prec 1$. Note that $\sin x$ and $\cos x$ are not related by any of these relations.

3 domination-type relations: $\succeq, \succ, \asymp$

Thus the triplet of notations $\succeq, \succ, \asymp$ for real functions correspond to the binary relations $\geq, >, =$ for real numbers.

Domination provides "implementation platform independence" for our complexity results: it does not matter whether you implement a given algorithm in a high level program language like `Java` or in assembly language. The complexity of your algorithm in these implementations (if done correctly) will be dominated by each other (i.e., same Θ-order). This also insulates our complexity results against Moore's Law: over a limited time period, the timing of our algorithms keeps the same Θ-order. Of course, Moore's law cannot hold indefinitely because of physical limits, but the end is not in sight yet.

one form of Moore's law predicts that the "speed" of hardware will double every 18 months...

¶30. The Big-Oh Notation. We write

$$\mathcal{O}(f)$$

(and read **order of** f or **big-Oh of** f) to denote the set of all complexity functions g such that

$$0 \preceq g \preceq f.$$

the key asymptotic notation to know!

Note that each function in $\mathcal{O}(f)$ dominates 0, *i.e.*, is eventually non-negative. Thus, restricted to functions that are eventually non-negative, the big-Oh notation (viewed as a binary relation) is equivalent to domination.

big-Oh is almost the same as domination

> *We can unroll the big-Oh notation as follows: To prove $g = \mathcal{O}(f)$, you need to show some $C > 0$ and x_0 such that for all $x \geq x_0$, if $g(x) =\downarrow$ and $f(x) =\downarrow$ then $0 \leq g(x) \leq Cf(x)$. Remember your epsilon-delta argument in Calculus? Well, the Computer Science analogue is the C-x_0 argument.*

$\delta : \epsilon :: C : x_0$

E.g., The set $\mathcal{O}(1)$ comprises all functions f that are bounded and eventually non-negative. The function $1 + \frac{1}{n}$ is a member of $\mathcal{O}(1)$.

The simplest usage of this $\mathcal{O}$-notation is as follows: we write

$$g = \mathcal{O}(f)$$

(and read 'g **is big-Oh of** f' or 'g **is order of** f') to mean g is a member of the set $\mathcal{O}(f)$. The equality symbol '=' here is "uni-directional": $g = \mathcal{O}(f)$ does not mean the same thing as $\mathcal{O}(f) = g$. Below, we will see how to interpret the latter expression. The equality symbol in this context is called a **one-way equality**. Why not just use '$\in$' for the one-way equality? Main reason is convenience: if $f \in O(g)$, $g \in O(h)$, $h = O(\cdots)$, and we want to merge these statements, then we need to write $f \in O(g) \subseteq O(h) \subseteq \cdots$. This is awkward. With one-way equalities, we simply write $f = O(g) = O(h) = \cdots$. The equality symbol has a uni-directional flavor whereby we (repeatedly) transform a formula from an unknown form into a known form, separated by an equality symbol. For example, the following sequence of one-way equalities

$$f(n) = \sum_{i=1}^{n}(i + \frac{n}{i}) = \left(\sum_{i=1}^{n} i\right) + \left(\sum_{i=1}^{n} \frac{n}{i}\right) = \mathcal{O}(n^2) + \mathcal{O}(n \log n) = \mathcal{O}(n^2)$$

is a derivation to show f is at most quadratic.

¶* 31. Big-Oh Expressions. The expression '$\mathcal{O}(f(n))$' is an example of an $\mathcal{O}$-expression, which we now define. In any $\mathcal{O}$-expression, there is a **designated variable** which is the real variable that goes[10] to infinity. For instance, the $\mathcal{O}$-expression $\mathcal{O}(n^k)$ would be ambiguous were it not for the tacit convention that 'n' is normally the designated variable. Hence k is assumed to be constant. We shall define $\mathcal{O}$**-expressions** as follows:

(Basis) If f is the symbol for a function, then f is an $\mathcal{O}$-expression. If n is the designated variable for $\mathcal{O}$-expressions and c a real constant, then both 'n' and 'c' are also $\mathcal{O}$-expressions.

(Induction) If E, F are $\mathcal{O}$-expressions and f is a symbol denoting a complexity function then the following are $\mathcal{O}$-expressions:

$$\mathcal{O}(E), \quad f(E), \quad E + F, \quad EF, \quad -E, \quad 1/E, \quad E^F.$$

Each $\mathcal{O}$-expression E denotes a set $\widetilde{E}$ of partial real functions in the obvious manner: in the basis case, a function symbol f denotes the singleton set $\widetilde{f} = \{f\}$. Inductively, the expression $E + F$ (for instance) denotes the set $\widetilde{E + F}$ of all functions $f + g$ where $f \in \widetilde{E}$ and $g \in \widetilde{F}$. Similarly for

$$\widetilde{f(E)}, \quad \widetilde{EF}, \quad \widetilde{-E}, \quad \widetilde{E^F}.$$

The set $\widetilde{1/E}$ is defined as $\left\{1/g : g \in \widetilde{E} \;\&\; 0 \preceq g\right\}$. The most interesting case is the expression $\mathcal{O}(E)$, called a "simple big-Oh expression". In this case,

$$\widetilde{\mathcal{O}(E)} = \left\{f : (\exists g \in \widetilde{E})[0 \preceq f \preceq g]\right\}.$$

Examples of $\mathcal{O}$-expressions:

$$2^n - \mathcal{O}(n^2 \log n), \qquad n^{n+\mathcal{O}(\log n)}, \qquad f(1 + \mathcal{O}(1/n)) - g(n).$$

Note that in general, the set of functions denoted by an $\mathcal{O}$-expression need not dominate 0. If E, F are two $\mathcal{O}$-expressions, we may write

$$E = F$$

[10] More generally, we can consider x approaching some other limit, such as 0.

to denote $\widetilde{E} \subseteq \widetilde{F}$, *i.e.*, the equality symbol stands for set inclusion! This generalizes our earlier "$f = \mathcal{O}(g)$" interpretation. Some examples of this usage:

$$\mathcal{O}(n^2) - 5^{\mathcal{O}(\log n)} = \mathcal{O}(n^{\log n}), \qquad n + (\log n)\mathcal{O}(\sqrt{n}) = n^{\log\log n}, \qquad 2^n = \mathcal{O}(1)^{n-\mathcal{O}(1)}.$$

An ambiguity arises from the fact that if $\mathcal{O}$ does not occur in an $\mathcal{O}$-expression, it is indistinguishable from an ordinary expression. We must be explicit about our intention, or else rely on the context in such cases. Normally, at least one side of the one-sided equation '$E = F$' contains an occurrence of '$\mathcal{O}$', in which case, the other side is automatically assumed to be an $\mathcal{O}$-expression. Some common $\mathcal{O}$-expressions are:

- $\mathcal{O}(0)$, the eventually zero functions.
- $\mathcal{O}(1)$, the bounded functions.
- $1 \pm \mathcal{O}(1/n)$, a set of functions that tends to $1^{\pm}$.
- $\mathcal{O}(n)$, the linearly bounded functions.
- $n^{\mathcal{O}(1)}$, the functions bounded by polynomials.
- $\mathcal{O}(1)^n$ or $2^{\mathcal{O}(n)}$, the functions bounded by simple exponentials.
- $\mathcal{O}(\log n)$, the functions bounded by some multiple of the logarithm.

¶* 32. Extensions of Big-Oh Notations. We note some simple extensions of the $\mathcal{O}$-notation:
(1) **Inequality interpretation:** For $\mathcal{O}$-expressions E, F, we may write $E \neq F$ to mean that the set of functions denoted by E is not contained in the set denoted by F. For instance, $f(n) \neq \mathcal{O}(n^2)$ means that for all $C > 0$, there are infinitely many n such that $f(n) > Cn^2$.
(2) **Subscripting convention:** We can subscript the big-Oh's in an $\mathcal{O}$-expression. For example,

$$\mathcal{O}_A(n), \quad \mathcal{O}_1(n^2) + \mathcal{O}_2(n \log n). \tag{19}$$

The intent is that each subscript (A, 1, 2) picks out a specific but anonymous function in (the set denoted by) the unsubscripted $\mathcal{O}$-notation. Furthermore, within a given context, two occurrences of an identically subscripted $\mathcal{O}$-notation are meant to refer to the same function. For subscripted expressions, it now makes sense to use inequalities, as in "$f \geq \mathcal{O}_A(g)$" or "$f \leq \mathcal{O}_1(g)$".

For instance, if A is a linear time algorithm, we may say that "A runs in time $\mathcal{O}_A(n)$" to indicate that the choice of the function $\mathcal{O}_A(n)$ depends on A. Further, all occurrences of "$\mathcal{O}_A(n)$" in the same discussion will refer to the same anonymous function. Again, we may write

$$n2^k = \mathcal{O}_k(n), \quad n2^k = \mathcal{O}_n(2^k)$$

depending on one's viewpoint. Especially useful is the ability to do "in-line calculations". As an example, we may write

$$g(n) = \mathcal{O}_1(n \log n) = \mathcal{O}_2(n^2)$$

where, it should be noted, the equalities here are true equalities of functions.

(3) Another possible extension is to multivariate real functions. For instance consider the notation "$f(x, y) = \mathcal{O}(g(x, y))$" where we view both x and y are designated variables. I.e., there exist constants $C > 0, x_0, y_0$ such that for all $x > x_0, y > y_0$, $f(x, y) \leq Cg(x, y)$. In practice, such an extension is seldom needed.

¶33. Other Asymptotic Notations: small-oh and super-domination So far, we have define a pair of closely related concepts: a *binary relation* called domination $f \succeq g$, and an *order notation* called Big-Oh $g = O(f)$. The difference between the two concepts is that the order notation has additional requirements, namely $g = O(f)$ must satisfy $g \succeq 0$. We now introduce four more pairs of such asymptotic notations. This is given by the second and last columns of following table:

Name of $\odot$	Notation $g = \odot(f)$	Informal Meaning	Definition of order $g = \odot(f)$	Analogous In-fix Notation
big-Oh	$g = O(f)$	$g \leq f$	$(\exists C > 0)[C \cdot f \geq g \geq 0(\text{ev.})]$	$f \succeq g$ (dominates)
big-Omega	$g = \Omega(f)$	$g \geq f$	$(\exists C > 0)[C \cdot g \geq f \geq 0(\text{ev.})]$	$f \preceq g$ (dominated by)
Theta	$g = \Theta(f)$	$g = f$	$(\exists C > 1)[C^2 \cdot f \geq C \cdot g \geq f \geq 0(\text{ev.})]$	$g \asymp f$ (co-dominates)
small-oh	$g = o(f)$	$g \ll f$	$(\forall C > 0)[C \cdot f > g \geq 0(\text{ev.})]$	$f \succ\!\!\succ g$ (super-dominates)
small-omega	$g = \omega(f)$	$g \gg f$	$(\forall C > 0)[C \cdot g > f \geq 0(\text{ev.})]$	$f \prec\!\!\prec g$ (super-dominated by)

The four new order notations (analogous to $O(f)$) are

$$\text{big-Omega } \Omega(f); \text{ Theta } \Theta(f); \text{ small-oh } o(f); \text{ small-omega } \omega(f).$$

Their formal definitions are given by Column 4 of the table. The first 3 rows of the table (big-Oh, big-Omega, Theta) are easily understood in terms of the domination relation, $f \succeq g$. But the last 2 rows require a binary relation called "super-domination". We say f **super-dominates** g, written $f \succ\!\!\succ g$ if

$$(\forall C > 0)[C \cdot f > g \text{ (ev.)}] \tag{20}$$

Note that we use $>$ instead of $\geq$ in (20). In our definition of domination $f \succ g$, we use $\geq$. In most applications, this subtlety change is irrelevant. However, our definition rules out "$0 \succ\!\!\succ 0$".

$\succ\!\!\succ$ *is like the informal concept* '$\gg$'

The corresponding set notations are called **small-oh/small-omega**, analogous to big-Oh/big-Omega. By definition, $f \succ\!\!\succ g$ iff $g \prec\!\!\prec f$. Note that $f \succ\!\!\succ g$ implies $f \succ g$.

Remark: The literature often define "$g = o(f)$" or "$f \succ\!\!\succ g$" by using limits, i.e., $g(x)/f(x) \to 0$ as $x \to \infty$. In the spirit of the non-calculus or "elementary approach", we simply avoid discussion of limits.

> CONVENTION: We observe that for any of the five choices of $\odot$, the set $\odot(f)$ is non-empty iff $f \succeq 0$. Henceforth, we adopt this convention: *whenever we write the expression "$\odot(f)$", it is assumed to be non-empty, i.e., $f \succeq 0$.*

We now unpack these definitions in a leisurely manner.

Big-Omega notation: $\Omega(f)$ is the set of all complexity functions g such that for some constant $C > 0$,

$$C \cdot g \geq f \geq 0 \text{ (ev.)}.$$

Of course, this can be compactly written as $g \succeq f \succeq 0$. Clearly, big-Omega is just the reverse of the big-Oh relation: $g = \Omega(f)$ iff $f = \mathcal{O}(g)$.

Theta notation: $\Theta(f)$ is the intersection of the sets $\mathcal{O}(f)$ and $\Omega(f)$. So g is in $\Theta(f)$ iff $g \asymp f$.

Small-oh notation: $o(f)$ is the set of all complexity functions g such that for all $C > 0$,

so C can be arbitrarily small!

$$C \cdot f > g \geq 0 \text{ (ev.)}.$$

As usual, we write $g = o(f)$ to mean $g \in o(f)$. For instance, with $f(x) = 1$ and $g(x) = 1/x$, we conclude that $1/x = o(1)$. Also, we have the relation $o(f) \subseteq \mathcal{O}(f)$.

Small-omega notation: $\omega(f)$ is the set of all functions g such that for all $C > 0$,

$$C \cdot g > f \geq 0 \text{ (ev.)}.$$

Clearly $\omega(f) \subseteq \Omega(f)$.

For each of these notations, we can again define the $\odot$-expressions ($\odot \in \{\Omega, \Theta, o, \omega\}$), use the one-way equality instead of set-membership or set-inclusion, and employ the subscripting convention. Thus, we write "$g = \Omega(f)$" instead of saying "g is in $\Omega(f)$". We call the set $\odot(f)$ the $\odot$-**order** of f. Here are some immediate relationships among these notations:

- $f = \mathcal{O}(g)$ iff $g = \Omega(f)$.
- $f = \Theta(g)$ iff $f = \mathcal{O}(g)$ and $f = \Omega(g)$.
- $f = \mathcal{O}(f)$ and $\mathcal{O}(\mathcal{O}(f)) = \mathcal{O}(f)$.
- $f + o(f) = \Theta(f)$.
- $o(f) \subseteq \mathcal{O}(f)$.
- $g = \omega(f)$ iff $f = o(g)$.

¶* 34. Varieties of Lower Bounds. It is instructive to explore the notions of a lower bound — one motivation is that lower bounds concepts are often misused in the literature. In the following, it is simpler if we assume that $f, g \geq 1$ (ev.). How can we express lower bounds on a complexity function f?

- One way is to say that g is a lower bound on f is $f = \Omega(g)$. This translates into $f = \Omega(g)$

$$(\exists C > 0)(\exists n_0)(\forall n > n_0)[Cf(n) > g(n)]. \tag{21}$$

- But we could also negate the upper bound statement $f = \mathcal{O}(g)$. Thus the statement $f \neq \mathcal{O}(g)$ gives another kind of lower bound on f: $f \neq \mathcal{O}(g)$

$$(\forall C > 0)(\forall n_0)(\exists n > n_0)[Cf(n) > g(n)]. \tag{22}$$

- Using the small-omega and small-oh notations, we have two other ways to state lower bounds. Thus $f = \omega(g)$ translates into $f = \omega(g)$

$$(\forall C > 0)(\exists n_0)(\forall n > n_0)[f(n) > Cg(n)]. \tag{23}$$

- And finally $f \neq o(g)$ translates into $f \neq o(g)$

$$(\exists C > 0)(\forall n_0)(\exists n > n_0)[f(n) > Cg(n)]. \tag{24}$$

Notice that the matrix '$[f(n) > Cg(n)]$' is common to all four lower bound statements (21)–(24). Of these, two are direct application of our notations ($= \Omega(g)$ and $\omega(g)$) but two are *negations* of our notations ($\neq \mathcal{O}(g)$ and $\neq o(g)$). It can be seen from the above translations that the four lower bounds are related by the following four implications:

$$\begin{array}{ccccc} & & f = \Omega(g) & & \\ & \nearrow & & \searrow & \\ f = \omega(g) & & & & f \neq o(g) \\ & \searrow & & \nearrow & \\ & & f \neq O(g) & & \end{array} \tag{25}$$

Likewise, we could introduce four ways of stating upper bounds.

Let us see how these notations are used in practice. For example, let us prove that for all $k < k'$,

$$n^{k'} \neq \mathcal{O}(n^k).$$

Suppose $n^{k'} = \mathcal{O}(n^k)$. Then there is a $C > 0$ such that $n^{k'} \leq Cn^k$ (ev.). That means $n^{k'-k} \leq C$ (ev.). This is a contradiction because n^ε is unbounded for any $\varepsilon > 0$.

¶* 35. What if Θ-order is too coarse? Despite our general aim of looking only at Θ-order of complexity functions, there is often need to refine this. One concept that is often used in the mathematical literature is this: write

$$f \sim g \tag{26}$$

if $f = g \pm o(g)$ or $f(x) = g(x)[1 \pm o(1)]$. This says that f and g approximate each other with relative error of $o(1)$.

so $n \sim n + \lg n$ but $n \not\sim 2n$.

Let us illustrate the need for this notation using the **unbounded search problem**: given an unknown real number x^*, we want to an interval $[a, b]$ containing x^* such that $b - a \leq 1$. When we have found such an interval, we say that x^* has been "located". We assume an **oracle for** x^*. We can pose queries of the form "is $x^* \leq x$?" (for any chosen $x \in \mathbb{R}$). The oracle will either answer yes or no. Based on this answer, we can pose another query, etc. How many queries must we ask before we can locate x^*? It is relatively to solve the unbounded search problem using

this is like the comparison tree model, but the tree is now infinite.

$$T_1(x^*) = 2\lg(|x^*|) + O(1) \tag{27}$$

queries. But we can also solve it with

$$T_2(x^*) = \lg(|x^*|) + 2\lg\lg(|x^*|) + O(1) \tag{28}$$

queries, or even better,

$$T_3(x^*) = \lg(|x^*|) + \lg\lg(|x^*|) + \lg\lg\lg(|x^*|) + O(1). \tag{29}$$

The problem of unbounded search will be taken up in Chapter 12. For now, we notice that $T_1 \not\sim T_2$, but $T_2 \sim T_3$. In order to distinguish T_3 from T_2, we propose the following definition: let

$$f \overset{+}{\preceq} g \iff 2^f \preceq 2^g \tag{30}$$

Similarly, $f \overset{+}{\asymp} g$ iff $f \overset{+}{\preceq} g$ and $g \overset{+}{\preceq} f$. Finally, $f \overset{+}{\prec} g$ iff $f \overset{+}{\preceq} g$ but not $g \overset{+}{\preceq} f$. Now, we see that

$$T_3 \overset{+}{\prec} T_2.$$

Call these $\overset{+}{\preceq}$, $\overset{+}{\prec}$, $\overset{+}{\asymp}$ the **+-domination** concepts. Intuitively, they provide approximations up to additive constants, in contrast to the standard domination concepts ($\preceq$, $\prec$, $\asymp$) which are approximations up to multiplicative constants.

¶* 36. Discussion of asymptotic notations. There is some debate over the best way to define the asymptotic concepts in computer science. So it is not surprising that there is considerable divergence on the details in the literature. Here we note just two alternatives: *be warned!*

- Perhaps the most common definition follows Knuth [4, p. 104] who defines "$g = \mathcal{O}(f)$" to mean there is some $C > 0$ such that $|f(x)|$ dominates $C|g(x)|$. Using this definition, both $\mathcal{O}(-f)$ and $-\mathcal{O}(f)$ would mean the same thing as $\mathcal{O}(f)$. But our definition allows us to distinguish[11] between $1+\mathcal{O}(1/n)$ and $1-\mathcal{O}(1/n)$. Note that $g = 1-\mathcal{O}(f)$ amounts to $1 - Cf \leq f \leq 1$ (ev.). When an big-Oh expression appears in negated form as in $-\mathcal{O}(1/n)$, it is really a lower bound
- Again, we could have defined "$\mathcal{O}(f)$" more simply, as comprising those g such that $g \preceq f$. That is, we omit the requirement $0 \preceq g$ from our original definition. This alternative definition is attractive for its simplicity. But the drawback of this simplified "$\mathcal{O}(f)$" is that it contains arbitrarily negative functions. The expression $1 - \mathcal{O}(1/n)$ is useful as an upper and lower bound under our official notation. But with the simplified definition, the expression "$1 - \mathcal{O}(1/n)$" has no value as an upper bound. Our official definition opted for something that is intermediate between this simplified version and Knuth's.

We are following Cormen et al [1] in restricting the elements of $\mathcal{O}(f)$ to complexity functions that dominate 0. This approach has its own burden: thus whenever we say "$g = \mathcal{O}(f)$", we have to check that g dominates 0 (cf. exercise 1 below). In practice, this requirement is not much of a burden, and is silently passed over.

A common abuse is to use big-Oh notations in conjunction with the inequality symbol ($\leq$). It is very tempting to write "$f(n) \leq \mathcal{O}(g)$" instead of the correct "$f(n) = \mathcal{O}(g)$". At best, this is redundant. The problem is that, once this notation is admitted, one may in the course of a long derivation eventually write "$f(n) \geq \mathcal{O}(E)$" which is not very meaningful. Hence we regard any use of $\leq$ or $\geq$ symbols in $\mathcal{O}$-notations as illegitimate (but this is legitimate again under the subscripting convention (19)).

Perhaps most confusion (and abuse) in the literature arises from variant definitions of the Ω-notation. For instance, one may have only shown a lower bound of the form $f \neq \mathcal{O}(g)$ or $f \neq o(g)$ result, but this is viewed as a proof of $f = \Omega(g)$ or $g = \omega(g)$. We see from (25) that these are quite different.

Evidently, these asymptotic notations can be intermixed. E.g., $o(n^{\mathcal{O}(\log n)} - \Omega(n)$. However, they can be tricky to understand and there seems to be little need for them. Another generalization with some applications are multivariate complexity functions such as $f(x, y)$. They do arise in discussing tradeoffs between two or more computational resources such as space-time, area-time, etc. In recently years, the study of "parametrized complexity" has given example of multivariate complexity functions where some of the size variables controls the "parameters" of the problem.

Exercises

Exercise 7.1: What is the relation between $A \equiv (P(x) \wedge Q(x))$ (i.o. x) and $B \equiv (P(x)$ (i.o. $x)) \wedge (Q(x)$ (i.o. $x))$ ♢

[11] On the other hand, there is no easy way to recover Knuth's definition using our definitions. It may be useful to retain Knuth's definition by introducing a special notation "$|\mathcal{O}|(f(n))$", etc.

Exercise 7.2: We defined the relation $f \succcurlyeq g$ as follows:

$$(\forall C > 0)[C \cdot f > g \text{ (ev.)}].$$

Suppose we modify it to be

$$(\forall C > 0)[C \cdot f \geq g \text{ (ev.)}].$$

Give an example of one relation $f \succcurlyeq g$ that holds under the new definition, but not in our official definition. ♢

Exercise 7.3: Below are two groups of 4 conditions each. Give the Hasse diagram of implications among conditions in each group. To show that your Hasse diagram has all possible relations, illustrated by functions that violate implications not in your Hasse diagram.
(A)

(A-i) $f > 0$ (ev.)
(A-ii) $f \succeq 0$
(A-iii) $f \succ 0$
(A-iv) $f \succcurlyeq 0$

(B) Give the Hasse diagram of implications among these conditions.

(B-i) $f > 1$ (ev.)
(B-ii) $f \succeq 1$
(B-iii) $f \succ 1$
(B-iv) $f \succcurlyeq 1$

♢

Exercise 7.4: (a) Suppose that for all $C > 0$, we have $f > Cg$ infinitely often (i.o.). Please express this using our asymptotic notations (like dominance, etc).
(b) Please restate the condition $f \not\prec g$ (f is not super-dominated by g) using the "infinitely often" terminology.
(c) Show two functions f, g such that $f \prec g$ but $f \not\prec g$. ♢

Exercise 7.5: Our asymptotic notations falls under two groups: O, Ω, Θ and o, ω. In the first group, we have $\Theta(f) = O(f) \cap \Omega(f)$. This suggests the "small-theta" analogue for the second group, "$\theta(f) = o(f) \cap \omega(f)$". Why was this not done? ♢

Exercise 7.6: Let $P : D \to \{0, 1\}$ be a partial predicate over some domain D. When we do quantification, $\forall x$ and $\exists x$ it is assumed that x range over D. Show that the following equivalences (called "de Morgan's laws for quantifiers") hold:
(a) $\neg(\forall x)P(x)$ is equivalent to $(\exists x)\neg P(x)$
(b) $\neg(\exists x)P(x)$ is equivalent to $(\forall x)\neg P(x)$ ♢

Exercise 7.7: To do this problem, we recall some common mathematical expressions:
(i) "f is unbounded" means that for any $C > 0$, there exists x such that $f(x) > C$.
(ii) "$f > g$ (i.o.)" (i.o. = infinitely often) means that there are arbitrarily large values of x where $f(x) > g(x)$ holds.
(iii) "f is bounded away from 0" means there exists $\epsilon > 0$ such that for all x, $f(x) \geq \epsilon$.

(a) Condition is "$f \nsucc 1$". Show an f that is unbounded but does not satisfy this condition.

(b) Condition is "$f \not\preceq 1$" (i.e., "It is not the case that $f \preceq 1$"). Give an English expression for this condition.

(c) Condition is "$f \succ 1$". Give an English expression for this condition.

(d) Clearly, Condition (a) implies Condition (b). Give a counter example for the converse.

♢

Exercise 7.8: Assume $f(n) \geq 1$ (ev.).

(a) Show that $f(n) = n^{\mathcal{O}(1)}$ iff there exists $k > 0$ such that $f(n) = \mathcal{O}(n^k)$. This is mainly an exercise in unraveling our notations!

(b) Show a counter example to (a) in case $f(n) \geq 1$ (ev.) is false. ♢

Exercise 7.9: Prove or disprove: $f = \mathcal{O}(1)^n$ iff $f = 2^{\mathcal{O}(n)}$. ♢

Exercise 7.10: If $P_i : D \to \{0, 1\}$ are partial predicates ($i = 0, 1$) over some domain D, then so are $\neg P_i$, $P_0 \vee P_1$ and $P_0 \wedge P_1$ where we use the rule that $\neg P_0(x), P_0(x) \vee P_1(x), P_0(x) \wedge P_1(x)$ are all undefined when $P_0(x) = \uparrow$. Show that $\neg(\forall x)P(x)$ is equivalent to $(\exists x)\neg P(x)$ and $\neg(\exists x)P(x)$ is equivalent to $(\forall x)\neg P(x)$. NOTE: these are called De Morgan's law for quantifiers, which is well-known when P is a total predicate. ♢

Exercise 7.11: Unravel the meaning of the $\mathcal{O}$-expression: $1 - \mathcal{O}(1/n) + \mathcal{O}(1/n^2) - \mathcal{O}(1/n^3)$. Does the $\mathcal{O}$-expression have any meaning if we extend this into an infinite expression with alternating signs? ♢

Exercise 7.12: For basic properties of the logarithm and exponential functions, refer to the Appendix in Chapter II. In the following, n is the designated variable, but c, k are constants. To prove a relation, you must explicitly specify the numerical constants (e.g., $n_0 = 3$ or $C = 2.5$) implicit in the asymptotic notations. *Try to use only elementary arguments.* Please prove the following:

(a) $(n + c)^k = \Theta(n^k)$. Note that c, k can be negative.

(b) $\log(n!) = \Theta(n \log n)$.

(c) $n! = o(n^n)$.

(d) $\lceil \log n \rceil ! = \Omega(n^k)$ for any $k > 0$.

(e) $\lceil \log \log n \rceil ! \leq n$ (ev.). ♢

Exercise 7.13: Show that $\ln(n!) = n \ln n + \Theta(n)$. You must not use Stirling's formula, but use the fact that the Harmonic numbers $H_n = \ln n + \Theta(1)$. ♢

Exercise 7.14: Provide either a counter-example when false or a proof when true. The base b of logarithms is arbitrary but fixed, and $b > 1$. Assume the functions f, g are arbitrary (do not assume that f and g are ≥ 0 eventually – the follow-up question next make assume additional properties of f, g.)

(a) $f = \mathcal{O}(g)$ implies $g = \mathcal{O}(f)$.

(b) $\max\{f, g\} = \Theta(f + g)$.

(c) If $g > 1$ and $f = \mathcal{O}(g)$ then $\ln f = \mathcal{O}(\ln g)$.
(d) $f = \mathcal{O}(g)$ implies $f \circ \log = \mathcal{O}(g \circ \log)$. Assume that $g \circ \log$ and $f \circ \log$ are complexity functions.
(e) $f = \mathcal{O}(g)$ implies $2^f = \mathcal{O}(2^g)$.
(f) $f = o(g)$ implies $2^f = \mathcal{O}(2^g)$.
(g) $f = \mathcal{O}(f^2)$.
(h) $f(n) = \Theta(f(n/2))$. ♢

Exercise 7.15: Re-solve the previous exercise, assuming that $f, g \geq 2$ (ev.) and $f + g =\downarrow$ (ev.). ♢

Exercise 7.16: Let $f(x) = \sin x$ and $g(x) = 1$.
(i) Prove $f \preceq g$ or its negation.
(ii) Prove $g \preceq f$ or its negation.
HINT: To prove that $f \not\preceq g$, you need to show that for *all* choices of $C > 0$ and $x_0 > 0$, some relationship between f and g fails. ♢

Exercise 7.17: Suppose $T_A(n)$ is the running time of an algorithm A. We consider some notions of lower bounds on $T_A(n)$:
(a) Suppose you have constructed an infinite sequence of inputs $I_1, I_2, \ldots$ of sizes $n_1 < n_2 < \cdots$ such that A on I_i takes time more than $f(n_i)$. How can you express this lower bound result using our asymptotic notations?
(b) In the spirit of (a), what would it take to prove a lower bound of the form $T_A(n) \neq \mathcal{O}(f(n))$? What must you show about of your constructed inputs $I_1, I_2, \ldots$.
(c) What does it take to prove a lower bound of the form $T_A(n) = \Omega(f(n))$?
(d) Can you think of other ways to express lower bounds? ♢

Exercise 7.18: Using our order notations $O(f), \Omega(f), o(f), \omega(f)$ to answer this question:
(i) State four ways to say that "f is an upper bound on g".
(ii) Give any logical relations among the four ways in (i), assuming that $g \geq 0$ (ev.). If there is no relation, give a counter example. ♢

Exercise 7.19: Show some examples where you might want to use "mixed" asymptotic expressions. ♢

Exercise 7.20: Suppose $P(x, y)$ is a partial predicate, and $Q(y)$ is $(\forall x)P(x, y)$. Using our definitions, $Q(y)$ is now a total predicate. Should we modify our treatment of quantifiers to allow $Q(y)$ to be a partial predicates? ♢

Exercise 7.21: Discuss the meaning of the expressions $n - \mathcal{O}(\log n)$ and $n + \mathcal{O}(\log n)$ under (1) our definition, (2) Knuth's definition and (3) the "simplified definition" in the discussion. ♢

End Exercises

§8. Conclusion: Two Dictums of Algorithmics

To conclude this overview of algorithmics, we state two principles in algorithmics. They justify many of our procedures and motivate some of the fundamental questions we ask.

¶37. *(A) Complexity functions are determined only up to Θ-order.* This recalls our motivation for introducing asymptotic notations, namely, concern for robust complexity results. For instance, we might prove a theorem that the running time $T(n)$ of an algorithm is "linear time", $T(n) = \Theta(n)$. Then simple and local modifications to the algorithm, or reasonable implementations on different platforms, should not affect the validity of this theorem.

There are many important caveats. We conclude from this dictum that it is important to design new algorithms with better Θ-complexity (such algorithms attain new "records" in the race towards optimality). While this attitude is good in itself, the converse attitude can be counter productive: we must not infer that only algorithms that achieve new records are important. Often, an asymptotically superior algorithm may be much slower than a slower algorithm when run on inputs of realistic sizes. For some problems, we might be interested in the constant multiplicative factors hidden by the Θ-notation. We also know that our ability to capture the simultaneous complexity of more than one computational resource is very limited. Finally, there are non-complexity issues that may matter. Simplicity of an algorithm is always appealing, in a non-quantifiable way. Ease-of-implementation might trump a purely complexity-based criterion. In short, we need a holistic view of algorithmics.

¶38. *(B) Problems with complexity that are polynomial-bounded are feasible. Moreover, there is an unbridgeable gap between polynomial-bounded problems and those that are not polynomial-bounded.* This principle goes back to Cobham and Edmonds in the late sixties and relates to the P versus NP question. Hence, the first question we ask concerning any problem is whether it is polynomially-bounded. The answer may depend on the particular complexity model. E.g., a problem may be polynomial-bounded in space-resource but not in time-resource, although at this moment it is unknown if this possibility can arise. Of course, polynomial-bounded complexity $T(n) = n^c$ is not practical except for small c (typically less than 6). In many applications, even $c = 2$ is not practical. So the "practically feasible class" is a rather small slice of P.

Despite caveats, these two dictums turn out to be extremely useful. The landscape of computational problems is thereby simplified and made "understandable". The quest for asymptotically good algorithms helps us understand the nature of the problem. Often, after a complicated but asymptotically good algorithm has been discovered, we find ways to achieve the same asymptotic result in a simpler (practical) way.

§9 APPENDIX A: General Notations

We gather some general notations used throughout this book. Use this as reference. If there is a notation you do not understand from elsewhere in the book, this is a first place to look.

Bookmark this appendix. Come back often!

¶A.0 Definitions.
We use the symbol $:=$ to indicate the definition of a term: we will write $X := \ldots Y \ldots$ when defining a term X in terms of $\ldots Y \ldots$. For example, we define the sign (sometimes called signum) function as follows:

$$\texttt{sign}(x) := \begin{cases} 1 & \text{iff } x > 0 \\ 0 & \text{iff } x = 0 \\ -1 & \text{iff } x < 0 \end{cases}$$

Or, to define the special symbol for logarithm to base 2, we write $\lg x := \log_2 x$.

¶A.1 Numbers.
Denote the set of natural numbers[12] by $\mathbb{N} = \{0, 1, 2, \ldots\}$, integers by $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$, rational numbers by $\mathbb{Q} = \{p/q : p, q \in \mathbb{Z}, q \neq 0\}$, the reals $\mathbb{R}$ and complex numbers $\mathbb{C}$. Thus we have

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}.$$

The positive and non-negative reals are denoted $\mathbb{R}_{>0}$ and $\mathbb{R}_{\geq 0}$, respectively. The set of integers $\{i, i+1, \ldots, j-1, j\}$ where $i, j \in \mathbb{N}$ is denoted $[i..j]$. So the size of $[i..j]$ is $\max\{0, j-i+1\}$. If r is a real number, let its **ceiling** $\lceil r \rceil$ be the smallest integer greater than or equal to r. Similarly, its **floor** $\lfloor r \rfloor$ is the largest integer less than or equal to r. Clearly, $\lfloor r \rfloor \leq r \leq \lceil r \rceil$. For instance, $\lfloor 0.5 \rfloor = 0$, $\lfloor -0.5 \rfloor = -1$ and $\lceil -2.3 \rceil = -2$. The **fractional part** of a number r is $r - \lfloor r \rfloor$ and is often denoted $\{r\}$.

For any positive integer m, let $\mathbb{Z}_m := \{0, 1, \ldots, m-1\}$. This set forms a ring in which we can do addition, subtraction and multiplication modulo m. When m is prime, it is even a field in which we can do division by any non-zero element.

¶A.2 Sets and Multisets.
The **size** or **cardinality** of a set S is the number of elements in S and denoted $|S|$. The empty set is $\emptyset$. A set of size one is called a **singleton**. The **disjoint union** of two sets is denoted $X \uplus Y$. This sometimes defined as

$$\{(x, 0) : x \in X\} \cup \{(y, 1) : y \in Y\}.$$

The intent to be able to uniquely attribute each element of $X \uplus Y$ to either X or Y. Of course, if $X \cap Y$ is empty, then there is no need for this artifice. We prefer to avoid this artifice and interpret $X \uplus Y$ as simply $X \cup Y$ with the side condition that $X \cap Y = \emptyset$. This allows us to say that if $z \in X \uplus Y$, then $z \in X$ or $z \in Y$, but not both. In particular, we may write $X = X_1 \uplus X_2 \uplus \cdots \uplus X_n$ to denote a **partition** of X into n subsets, i.e., the X_i's are pairwise disjoint and their union is X. If X is a set, then 2^X denotes the set of all subsets of X. The **Cartesian product** $X_1 \times \cdots \times X_n$ of the sets $X_1, \ldots, X_n$ is the set of all n-tuples of the form $(x_1, \ldots, x_n)$ where $x_i \in X_i$. If $X_1 = \cdots = X_n$ then we simply write this as X^n. If $n \in \mathbb{N}$ then an **n-set** refers to a set with cardinality n, and $\binom{X}{n}$ denotes the set of n-subsets of X.

[12] Zero is considered natural here, although the ancients do not consider it so. The symbol $\mathbb{Z}$ comes from the German word 'Zahl' for numbers or 'zahlen' to count.

Sometimes, we need to consider **multisets**. These are sets whose elements need not be distinct. E.g., the multiset $S = \{a, a, b, c, c, c\}$ has 6 elements but only three of them are distinct. There are two copies of a and three copies of c in S. Note that S is distinct from the set $\{a, b, c\}$. Since we use the usual set braces to write multisets, the notation is ambiguous unless we explicitly say whether we are discussing sets or multisets. Alternatively, a multiset can be viewed as a function $\mu : S \to \mathbb{N}$ whose domain is a standard set S. Intuitively, $\mu(a)$ is the multiplicity of each $a \in S$. Of course, we could further extend to $\mu : S \to \mathbb{R}_{\geq 0}$, etc.

¶A.3 Relations and Order.
An n-ary relation on a set X is a set of the form $R \subseteq X^n$. The most important case is $n = 2$, when we have binary relations. Instead of saying $(a, b) \in R$, we like to write aRb, read as "a is R-related to b".

Let $a, b, c \in X$. A binary relation R is **reflexive** if aRa, **transitive** if aRb and bRc implies aRc, **symmetric** if aRb implies bRa, **anti-symmetric** if aRb and bRa implies $a = b$. A **pre-order** R is a reflexive and transitive binary relation. A pre-order R that is also **symmetric** is an **equivalence** relation. Equivalence relations are an extremely important concept in all of mathematics, and it induces a partition of X into disjoint subsets, called equivalence classes. A pre-order R that is anti-symmetric is a **partial order** relation.

Lemma 3 *Let $R \subseteq X^2$ be a pre-order.*
(i) The set $\overline{X} := \{\overline{x} : x \in X\}$ where $\overline{x} = \{y \in X : xRy, yRx\}$ forms a partition of X.
(ii) The relation $\overline{R} \subseteq \overline{X}^2$ where $\overline{x}\overline{R}\overline{y}$ if xRy is a partial order on $\overline{X}$.

Proof. (i) Suppose $\overline{x} \cap \overline{y}$ is non-empty for some $x, y \in X$. Then there is a $z \in \overline{x} \cap \overline{y}$. We prove that $\overline{x} \subseteq \overline{y}$: for $u \in \overline{x}$ implies uRx. But xRz and zRy, so by transitivity, $uRxRzRy$ or uRy. We can similarly show yRu. Thus $u \in \overline{y}$. This proves $\overline{x} \subseteq \overline{y}$. Again by symmetry, we can show that $\overline{y} \subseteq \overline{x}$. Thus $\overline{x} = \overline{y}$. This proves that the sets $\overline{x}$ in $\overline{X}$ are pairwise disjoint. Moreover, every $x \in X$ belongs to $\overline{x} \in \overline{X}$. This concludes our proof that $\overline{X}$ is a partition of X.
(ii) We must prove reflexivity, antisymmetry and transitivity of $\overline{R}$. Reflexivity comes from $\overline{x}\overline{R}\overline{x}$ since xRx holds in a pre-order. Antisymmetry comes from $\overline{x}\overline{R}\overline{y}$ and $\overline{y}\overline{R}\overline{x}$ implies $y \in \overline{x}$ and hence $\overline{y} = \overline{x}$. Transitivity of $\overline{R}$ follows easily from the transitivity of R. **Q.E.D.**

A very useful concept in analysis of comparison based algorithms in the concept of a "Hasse Diagram". Fix a partial order $R \subseteq X^2$ on the set X. For $x, y \in X$, we say the relationship "$x > y$" holds if $(x, y) \in R$, etc. We say x **covers** y (relative to R) if $x > y$ and there is no z such that $x > z > y$. The **Hasse diagram** of R is a directed graph $H(R)$ whose vertex set is X and edges are $x-y$ whenever x covers y.

after Helmut Hasse (1898-1979)

CONVENTION: In figures, we will draw the digraph $H(R)$ such that x appears at a higher level than y, so the direction of edge is implicitly from higher to lower.

If R had been deduced from an algorithm based on comparisons on X, then the comparison $x : y$ must have been made by the algorithm. Relative to R, we call such a comparison **essential**.

¶A.4 Functions.
If $f : X \to Y$ is a partial function, then write $f(x) =\uparrow$ if $f(x)$ is undefined and $f(x) =\downarrow$ otherwise. If for all x, $f(x)\downarrow$, then f is a **total function**. Some authors use "$f : X \dashrightarrow Y$" to

indicate partial functions, and reserve "$f : X \to Y$" for total functions. Function composition will be denoted $f \circ g : X \to Z$ where $g : X \to Y$ and $f : Y \to Z$. Thus $(f \circ g)(x) = f(g(x))$. We need the special rule that when $g(x) = \uparrow$ then $f(g(x)) = \uparrow$. We say a total function f is **injective** or "1 to 1" if $f(x) = f(y)$ implies $x = y$; it is **surjective** or **onto** if $Y = \{f(x) : x \in X\}$; it is **bijective** if it is both injective and surjective.

The special functions of exponentiation $\exp_b(x)$ and logarithm $\log_b(x)$ to base $b > 0$ are more fully described in the Appendix of Chapter 2. Although these functions can be viewed as complex functions, we will exclusively treat them as real functions in this book. In particular, it means $\log_b(x)$ is undefined for $x \leq 0$. When the base b is not explicitly specified, it is assumed to be some constant $b > 1$. Two special bases[13] deserve their own notations: $\lg x$ and $\ln x$ refer to logarithms to base $b = 2$ and base $b = e = 2.718...$, respectively. In computer science, $\lg x$ is immensely useful. For any real a, we write $\log^a x$ as shorthand for $(\log x)^a$. E.g., $\log^2 x = (\log x)^2$. For any natural number i, let $\log^{(i)} x$ denote the i-fold application of the log-function. E.g., $\log^{(2)} x = \log(\log x)) = \log\log x$ and $\log^{(0)} x = x$. In fact, this notation can be extended to any integer i, where $i < 0$ indicates the $|i|$-fold application of exp.

¶A.5 Logic.

We assume the student is familiar with Boolean (or propositional) logic. In Boolean logic, each variable A, B stands for a proposition that is either true or false. Boolean logic deals with Boolean combinations of such variables: $\neg A, A \vee B, A \wedge B$. We also let $A \Rightarrow B$ denote **logical implication**, defined as $\neg A \vee B$. In particular $A \Rightarrow B$ iff $\neg B \Rightarrow \neg A$.

But mathematical facts go beyond propositional logic. Here is an example[14] of a mathematical assertion $P(x, y)$ about the real variables x, y:

$$P(x, y): \text{ There exists a real } z \text{ such that either } x \geq y \text{ or } x < z < y. \tag{31}$$

Assertions contain variables: for example, $P(x, y)$ in (31) contains x, y, z. Each variable has a range, often implicit: the variables x, y, z range over real numbers. Each variable is either **quantified** (either by "for all" or "there exists") or **unquantified**. Alternatively, quantified (unquantified) variables are said to be **bound** (**free**). In our example $P(x, y)$, z is bound while x, y are free. It is conventional to display the free variables as functional parameters of an assertion. The symbol $\forall$ stands for "for all" and is called the **universal quantifier**. Likewise, the symbol $\exists$ stands for "there exists" and is called the **existential quantifier**. Assertions with no free variables are called **statements**. We can always convert an assertion into a statement by adding some prefix to quantify each of the free variables. Thus, $P(x, y)$ can be converted into statements such as in (33) or as in $(\exists x \in \mathbb{R})(\forall y \in \mathbb{R})[P(x, y)]$. In general, if A and B are statements, so are any Boolean combinations of A and B, such as $A \wedge B$ and $\neg A$ or $A \vee B$. However, all statements can be transformed into the form

In computer programs, bound variables are just local variables and free variables are global variables!

$$(Q_1 x_1)(Q_2 x_2) \cdots (Q_n x_n)\,[P(x_1, x_2, \ldots, x_n)] \tag{32}$$

where Q_i is the ith quantifier part, each x_i is a distinct variable and $P(x_1, \ldots, x_n)$ a predicate. Such a form, where all the quantifiers appear before the predicate part, is said to be in **prenex form**.

The assertion $P(x, y)$ in (31) happens to be **valid**, meaning it is true for all instantiations of the free variables x, y. That is, if we bound all the free variables in $P(x, y)$ by universal quantifiers, then we get a true statement.

$$(\forall x, y \in \mathbb{R})[P(x, y)]. \tag{33}$$

[13] Of course $\ln x$ has the (well-deserved) appellation "natural logarithm", but $\lg x$ has no special name. I propose to to call it the "computer science logarithm".

[14] When we formalize the logical language of discussion, what is called "assertion" here is often called "formula".

All mathematical assertions are of this nature, i.e., valid. It is said that mathematical truths are universal: truthhood does not allow exceptions. If an assertion $P(x, y)$ has exceptions, let $E(x, y)$ be an assertion that holds at all exceptional instances of x, y (but $E(x, y)$ might capture more than just the exceptional instances). Now, the new statement $P(x, y) \vee E(x, y)$ is valid. A trivial choice for $E(x, y)$ is $\neg P(x, y)$.

In the above discussion, we make the conventional assumption that when the variables in an assertion are instantiated, then the assertion is either true or false. But in discussing partial functions, we need to generalize this to the setting where for some instances of x, y, the assertion $P(x, y)$ might be undefined (neither true nor false). We call P a **partial assertion** (or partial predicate). The quantified form $(\forall x)P(x)$ is then true if for all x in the domain, either $P(x)$ is undefined or $P(x)$ is true; similarly, $(\exists x)P(x)$ is true if there is some x in the domain such that $P(x)$ is defined and true. This extends naturally to predicates with more than one free variable.

More generally, we can define **predicates** to refer to any function with a finite domain. The usual 2-valued predicates may be called **logical predicates**. In geometric computation, we prefer a three-valued predicate since most geometric and topological distinctions are based on signs of real values.

¶A.6 Proofs and Induction.
Constructing proofs or providing counter examples to mathematical statements is a basic skill to cultivate. Three kinds of proofs are widely used: (i) case analysis, (ii) induction, and (iii) contradiction.

A proof by case analysis is often a matter of patience. But sometimes a straightforward enumeration of the possibilities will yield too many cases; clever insights may be needed to compress the argument. Induction is sometimes mechanical as well but very complicated inductions may also arise (Chapter 2 treats induction). Proof by contradiction may have a creative element: formulating an assertion to be contradicted!

> HINT: It is a good idea to indicate the assertion that is going to be contradicted. For instance, to prove that an integer function $f(n)$ satisfies $f(n) < n$ using a proof by contradiction, you may say "By way of contradiction, assume $f(n) \geq n$". If this is a mouthful, we suggest the short hand "BWOC, let $f(n) \geq n$". Use BWOC like the other beloved acronym WLOG in proofs.

OK, LOL, IMHO, DWNAA? Just made that one up, "Do we need another acronym?"

In proofs by contradiction, you will need to routinely negate a logical statement. Let us first consider the simple case of propositional logic. Here, you basically apply what is called De Morgan's Law: if A are B are truth values, then $\neg(A \vee B) = (\neg A) \wedge (\neg B)$ and $\neg(A \wedge B) = (\neg A) \vee (\neg B)$. For instance suppose you want to contradict the proposition $A \Rightarrow B$. You need to first know that $A \Rightarrow B$ is the same as $(\neg A) \vee B$. Negating this by de Morgan's law gives us $A \wedge (\neg B)$.

Next consider the case of quantified logic. **De Morgan's law for quantifiers** is the following pair of logical equivalences

$$\begin{aligned} \neg(\forall x)P(x) &\equiv (\exists x)[\neg P(x)], \\ \neg(\exists x)P(x) &\equiv (\forall x)[\neg P(x)]. \end{aligned}$$

In other words, we can push a negation past a quantifier by flipping the type of the quantifier. Of course, if P itself is quantified in prenex form, we can further transform $\neg P(x)$ by repeated

application of the rule: e.g.,

$$\neg(\forall x(\exists y)(\forall z)[P(x,y,z)] \equiv (\exists x)(\forall y(\exists z)[\neg P(x,y,z)]$$

A useful extension of De Morgan's law for quantifiers arise as follows: in most applications, we do not write a "bald" quantification (Qx) where Q is $\exists$ or $\forall$. Instant we "bound" the domain of x within the quantifier by writing "$(Qx \in D)$" for some set D. If x is a real number, we may bound x by expressions like $(Qx > 3.14)$ or $(Qx \leq x_0)$. Call $(Qx \in D)$ a **bounded quantifier**. It's interpretation is given by this translation:

$$(Qx \in D)[P(x)] \equiv (Qx)[x \in D \Rightarrow P(x)]$$

We then have the bounded form of De Morgan's law for quantifiers:

$$\begin{aligned} \neg(\forall x \in D)P(x) &\equiv (\exists x \in D)[\neg P(x)], \\ \neg(\exists x \in D)P(x) &\equiv (\forall x \in D)[\neg P(x)]. \end{aligned}$$

These laws remain valid even when P is a partial predicate. A useful place to exercise these rules is to do some proofs involving the asymptotic notation (big-Oh, big-Omega, etc). See Exercises.

A proof Π can be organized in a variety of ways, but perhaps the simplest format is a sequence of assertions, $\Pi = (A_1, A_2, \ldots, A_n)$ where each A_i is either known to be true or can be deduced from $A_1, \ldots, A_{i-1}$ sound rules of deduction. We can indicate this progression as

$$((\cdots((\texttt{true} \Rightarrow A_1) \Rightarrow A_2) \Rightarrow \cdots \Rightarrow A_{n-1}) \Rightarrow A_n)$$

where '$\Rightarrow$' should be read as 'implies'. We usually regard A_n as the conclusion of the proof. This is the normal direction of proof, where we proceed from known to new or unknown assertions. Of course, this is probably not the order in which you discover the proof. The more natural direction is the reverse direction, writing

$$(A_n \Leftarrow (A_{n-1} \Leftarrow \cdots \Leftarrow (A_n \Leftarrow (A_1 \Leftarrow \texttt{true}))\cdots))$$

where '$\Leftarrow$' should be read as 'provided'. The advantage of this reverse mode is that you begin from what is known or required, and reduce it to (hopefully) more elementary assertions to be verified. We illustrate this approach in §II.13 (orders of growth).

¶A.7 Formal Languages.
An **alphabet** is a finite set Σ of symbols. A finite sequence $w = x_1 x_2 \cdots x_n$ of symbols from Σ is called a **word** or **string** over Σ; the **length** of this string is n and denoted[15] $|w|$. When $n = 0$, this is called the **empty string** or **word** and denoted with the special symbol ϵ. The set of all strings over Σ is denoted Σ^*. A **language** over Σ is a subset of Σ^*.

¶A.8 Graphs.
A **hypergraph** is a pair $G = (V, E)$ where V is any set and $E \subseteq 2^V$. We call elements of V **vertices** and elements of E **hyper-edges**. In case $E \subseteq \binom{V}{k}$, we call G a k-graph. The case $k = 2$ is important and is called a **bigraph** (or more commonly, **undirected graph**). A **digraph** or **directed graph** is $G = (V, E)$ where $E \subseteq V^2 = V \times V$. For any digraph $G = (V, E)$, its **reverse** is the digraph (V, E') where $(u, v) \in E$ iff $(v, u) \in E'$. In this book, the word "graph" shall refer to a bigraph or digraph; the context should make the intent clear. The edges of graphs are often written '(u, v)' or 'uv' where u, v are vertices. We will prefer[16] to denote edge-hood by the notation $u{-}v$. Of course, in the case of bigraphs, $u{-}v = v{-}u$.

[15] This notation should not be confused with the absolute value of a number or the size of a set. The context will make this clear.

[16] When we write $u{-}v$, it is really an assertion that the (u, v) is an edge. So it is redundant to say "$u{-}v$ is an edge".

Often a graph $G = (V, E)$ comes with auxiliary data, say d_1, d_2, etc. In this case we denote the graph by

$$G = (V, E; d_1, d_2, \ldots)$$

using the semi-colon to mark the presence of auxiliary data. For example:
(i) Often one or two vertices in V are distinguished. If $s, t \in V$ are distinguished, we might write $G = (V, E; s, t)$. This notation might be used in shortest path problems where s is the source and t is the target for the class of paths under consideration.
(ii) A "weight" function $W : E \to \mathbb{R}$, and we denote the corresponding weighted graph by $G = (V, E; W)$.
(iii) Another kind of auxiliary data is **vertex coloring** of G, i.e., a function $C : V \to S$ where S is any set. Then $C(v)$ is called the **color** of $v \in V$. If $|S| = k$, we call C a k-coloring. The **chromatic graph** is therefore given by the triple $G = (V, E; C)$. An **edge coloring** is similarly defined, $C : E \to S$.

We introduce terminology for some special graphs: If V is the empty set, A graph $G = (V, E)$ is called the **empty graph**. If E is the empty set, $G = (V, E)$ is called the **trivial graph**. Hence empty graphs are necessarily trivial but not vice-versa. $K_n = (V, \binom{V}{2})$ denotes the **complete graph** on $n = |V|$ vertices. A **bipartite graph** $G = (V, E)$ is a digraph such that $E \subseteq V_1 \times V_2$ where $V = V_1 \uplus V_2$. It is common to write $G = (V_1, V_2, E)$ in this case. The special case $K_{m,n} := (V_1, V_2, V_1 \times V_2)$ is called the **complete bipartite graph** where $m = |V_1|$ and $n = |V_2|$. We also say that a bigraph G is **bipartite** when G does not contain an odd cycle.

Two graphs $G = (V, E), G' = (V', E')$ are **isomorphic** if there is some bijection $\phi : V \to V'$ such that $\phi(E) = E'$ (the notation $\phi(E)$ has the obvious meaning).

If $G = (V, E), G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ then we call G' a **subgraph** of G. In case E' is the restriction of E to the edges in V', *i.e.*, $E' = E \cap V' \times V'$, then we say G' is the subgraph of G **induced by** V', or G' is the **restriction** of G to V'. We may write $G|V'$ for G'.

A **path** (from v_1 to v_k) is a sequence $(v_1, v_2, \ldots, v_k)$ of vertices such that (v_i, v_{i+1}) is an edge. Thus, we may also denote this path as $(v_1 - v_2 - \cdots - v_k)$. Call v_1 and v_k the **start** and **terminus** of the path, or collectively, they are the **endpoints** of the graph. A path is **closed** if $v_1 = v_k$ and $k > 1$. Two closed paths are **cyclic equivalent** if the sequence of edges they pass through are the same up to cyclic reordering. A cyclic equivalence class of closed paths is called a **cycle**. The **length** of a path is one less than the vertices in sequence defining the path; the length of a cycle is just the length of any of its representative closed paths. A graph is **acyclic** if it has no cycles. Sometimes acyclic bigraphs are called **forests**, and acyclic digraphs are called **dags** ("directed acyclic graph"). *For bigraphs, we further require that that closed paths have the form* $(v_1 - v_2 - v_3 - \cdots - v_k)$ *where* $v_1 = v_k$ *and* $v_{i-1} \neq v_{i+1}$ *for all* $i = 1, \ldots, k$. Without this requirement, each edge $u - v$ in a bigraph would rise to a closed path of the form (u, v, u); such a bigraph would be considered "cyclic" for the wrong reason.

Two vertices u, v are **connected** if there is a path from u to v, and a path from v to u. (Note that in the case of bigraphs, there is a path from u to v iff there is a path from v to u.) We shall say v **is adjacent to** u if $u - v$. Connectivity is a symmetric binary relation for all graphs; adjacency is also a symmetric binary relation for bigraphs. It is easily seen that connectivity is also reflexive and transitive. This relation partitions the set of vertices into **connected components**.

In a digraph, **out-degree** and **in-degree** of a vertex is the number of edges issuing (respectively) from and into that vertex. The **out-degree** (resp., **in-degree**) of a digraph is the maximum of the out-degrees (resp., in-degrees) of its vertices. The vertices of out-degree 0 are

called **sinks** and the vertices of in-degree 0 are called **sources**. The **degree** of a vertex in a bigraph is the number of adjacent vertices; the **degree** of a bigraph is the maximum of degrees of its vertices.

See Chapter 4 for more graph concepts.

¶A.9 Trees.
A connected acyclic bigraph is called a **free tree**. A digraph such that there is a unique source vertex (called the **root**) and all the other vertices have in-degree 1, is called[17] a **tree**. The sinks in a tree are called **leaves** or **external nodes** and non-leaves are called **internal nodes**. In general, we prefer a terminology in which the vertices of trees are called **nodes**. Thus there is a unique path from the root to each node in a tree. If u, v are nodes in T then u is a **descendant** of v if there is a path from v to u. Every node v is a descendant of itself, called the **improper descendant** of v. All other descendants of v are called **proper**. We may speak of the **child** or **grandchild** of any node in the obvious manner. The **degree** of v is the number of children of v. The **degree** of a tree is the maximum degree of any of its nodes. The reverse of the descendant binary relation is the **ancestor** relation; thus we have **proper ancestors**, **parent** and **grandparent** of a node.

The **subtree** at any node u of T is the subgraph of T obtained by restricting to the descendants of u. The **size** of T is the number of nodes in T. The **depth** of a node u in a tree T is the length of the path from the root to u. So the root is the unique node of depth 0. The **depth of** T is the maximum depth of a node in T. The **height** of a node u is just the depth of the subtree at u; alternatively, it is the length of the longest path from u to its descendants. Thus u has height 0 iff u is a leaf iff u has no children. The collection of all nodes at depth i is also called the i**th level** of the tree. Thus level zero is comprised of just the root. We normally draw a tree with the root at the top of the figure, and edges are implicitly direction from top to bottom.

See Chapter 3 for more information on binary search trees.

¶A.10 Programs.
In this book, we present algorithms in an informal unspecified programming language that combines mathematical notations with standard programming language constructs. For lack of better name, we call this language **pseudo-PL**. The basic goal in the presentation of pseudo-PL programs is to expose the underlying algorithmic logic. It is not to produce code that can compile in any conventional programming language! And yet, it is often easy to transcribe pseudo-PL into compilable code in languages such as `C++` or `Java`. There are two good reasons why we stop short of writing compilable code – first, it is easier to understand, and second, it is programming language-dependent.

pseudo-PL is appropriately amorphous by design

[17] One can also define trees in which the sense of the edges are reversed: the root is a sink and all the leaves are sources. We will often go back and forth between these two view points without much warning. E.g., we might speak of the "path from a node to the root". While it is clear what is meant here, but to be technically correct, we ought to speak awkwardly of the path in the "reverse of the tree".

> Programming languages are harder to understand because they are intended for machine consumption, and that could get in the way of human understanding. A major advantage of writing compilable code is that it could be given to a computer for execution. Unfortunately, the "half-life" of programming languages tend to be rather short compared to that of natural languages. Informally, say the half-life of a programming language is the average time it takes before most programs in the language will no longer compile; similarly, the half-life of a natural language or pseudo code is the average time it takes before most people find hard to understand algorithmic descriptions. I would guess the former half-life at 1 year, and the latter half-life at 50 years. Therein lies the advantage of pseudo code.

Here is the quick run-down on pseudo-PL:

no clutter language

- We use standard programming constructs such as if-then-else, while-loop, return statements, etc.
- To reduce clutter, we indicate the structure of programming blocks by indentation and newlines only. In particular, we avoid explicit block markers such as "begin...end", "{...}", etc.
- Single line comments in a program are indicated in two ways:
  ▷ *This is a forward comment*
  ◁ *This is a backward comment*
  These comments either precede (if a forward comment) or follows (if a backward comment) the code that it describes. We have little need for multiline comments in pseudo-PL because the code is supplemented by off-line explanations to the same purpose.
- Programming variables are undeclared, and implicitly introduced through their first use. They are not explicitly typed, but the context should make this clear. This is in the spirit of modern scripting languages such as `Perl`, and consistent with our clutter-free spirit.
- Normally, each line is a command, so we may terminate a command line with the traditional semicolon (;) or without any punctuation marks if it is clear. If the line is more "Englishy", we prefer to terminate in a full stop. But if we put two or commands on one line, we generally separate them with semicolons. What if a command needs more than one line? In many computer languages, the continuation symbol is \. But in order to produce more human friendly programs, we could use ellipsis "..." at the end of a line to indicate its continuation to the next line. But if the line is an English sentence, we can even drop the ellipsis and indent the continuation line appropriately.

Programmers use '=' for assignment and '==' for equality test. We opt to preserve the mathematical meaning of '='.

- Informally, the equality symbol '=' is often overloaded to indicate the assignment operator as well as the equality test. We will use ← for assignment operator, and preserve '=' for equality test.
- In the style of `C` or `Java`, we write "x`++`" (resp., "`++`x") to indicate the increment of an integer variable x. The value of this expression is the value of x before (resp., after) incrementing. There is an analogous notation for decrementing, x`--` and `--`x.

Here is a recursive program written in pseudo-PL to compute the Factorial function:

```
FAC(n):
 Input: natural number n.
 Output: n!
 ▷ Base Case
1.    If n ≤ 1 Return(1)
 ▷ General Case
2.    Return(n · FAC(n − 1))    ◁ This is a recursive call
```

¶A.11 How to answer algorithmic exercises.

In our exercises, whenever we ask you to give an algorithm, it is best to write in pseudo code. We suggest you emulate our pseudo-PL form of presentation. Students invariably ask about what level of detail is sufficient. The general answer is *as much detail as one needs to know how to reduce it to compilable programs in a conventional programming language*. Here is a checklist you can use:

- *Specify your input and output.* This cannot be emphasized enough. We cannot judge your algorithm if we do not know what to expect from its output! *sine qua non!*
- *Take advantage of well-known algorithms.* For instance, if you need to to sort, you should generally be able to just[18] invoke a suitable sorting routine.
- *Reduce all operations to $\mathcal{O}(1)$ time operations.* Do this when Rule 1 does not apply. Sometimes, achieving $\mathcal{O}(1)$ time may depend on a suitable choice of data structures. If so, be sure to explain this.
- *Use progressive algorithm development.* Even pseudo code may be incomprehensible without a suitable orientation – it is never wrong to precede your pseudo code with some English explanation of what the basic idea is. In more complicated situations, do this in 3 steps: explain basic ideas, give pseudo code, further explain certain details in the pseudo code.
- *Use standard algorithmic paradigms.* In this book, we will see well-known paradigms such as divide-and-conquer, greedy methods, dynamic programming, etc. Another important paradigm is the notion of "shell-programming" i.e., driver programs; See tree and graph traversals in Chapters III and IV. *NOT in sense of bash, csh, tcsh, etc.*
- *Explain and initialize all variables and data structures.* Most non-trivial algorithms have some data structures, possibly the humble array. Critical variables (counters, coloring schemes) ought to be explained too. You must show how to initialize them.
- *The control structure of your algorithms should be evident.* All the algorithms you design should have simple control structures – typically a simple loop or a doubly-nested loops. Triply-nested loops do arise (e.g., dynamic programming) but deeper nesting is seldom needed. Each loop should use standard programming constructs (for-loop, while-loop, do-loop, etc). It is an axiom[19] that if a problem can be solved, then it is solvable by clean loop structures.
- *Correctness.* This is an implicit requirement of all algorithms. All the algorithms we study must halts on all (valid) inputs. Correctness of such algorithms is traditionally

[18] In computing, this is known as "code reuse". Others call this "not reinventing the wheel".

[19] There are theorems about the universality of loop-programs (Meyer and McCreight) and the possibility of avoiding "go-to" statements.

split into two distinct requirements:
(1) The algorithm halts.
(2) The output is correct when it halts. This is also known as **partial correctness**.
A simple method to prove partial correctness is this: at the beginning of each iteration of a loop, you should be able to attach a suitable **invariant** (called **assertion** in standard programming languages). Partial correctness follows easily if the appropriate invariants hold.

meta principle: *every loop has an invariant – you just have to look for it*

- *Analysis and Efficiency.* This is a more advanced requirement. But since this is what algorithmics is about, we view it as part and parcel of any algorithm in this book. You should always be able to give a big-Oh analysis of your algorithm. In most cases, non-polynomial time solutions are unnecessarily inefficient.

¶A.12 LIST OF ABBREVIATIONS

- AVL: Adel'son-Vel'skii and Landis tree (§III.6)
- BFS: Breadth First Search (§IV.4)
- BST: Binary search Tree (§III.3)
- DFS: Depth First Search (§IV.5)
- `Grd`: Greedy Algorithm for linear bin packing
- `FF`: First Fit Algorithm for bin packing
- iff: if and only if
- MST: Minimum Spanning Tree
- TSP: Traveling Salesman Problem
- WLOG or wlog: Without loss of generality

Exercises

Exercise 9.1: Please write a piece of Java code that has complexity $O(m(n^2+p))$ where m, n, p are positive integers. ♢

Exercise 9.2: Above we defined what it means to be a bipartite digraph and bipartite bigraph. Give the connection between the two concepts. ♢

Exercise 9.3: Show that if x is real and a a positive integer, then $\lfloor x/a \rfloor = \lfloor \lfloor x \rfloor /a \rfloor$. Similarly, $\lceil x/a \rceil = \lceil \lceil x \rceil /a \rceil$. ♢

Exercise 9.4: The following is a useful result about iterated floors and ceilings.

(a) Let $x > 0$ be real and $b > 1$ be an integer. Let $N_0 := \lfloor x \rfloor$ and for $i \geq 0$, $N_{i+1} := \lfloor N_i/b \rfloor$. Show that $N_i = \lfloor n/b^i \rfloor$. Similarly for ceilings.

(b) Let $u_0 = 1$ and $u_{i+1} = \lfloor 5u_i/2 \rfloor$ for $i \geq 0$. Show that for $i \geq 4$, $0.76(5/2)^i < u_i \leq 0.768(5/2)^i$. HINT: $r_i := u_i(2/5)^i$ is non-increasing; give a lower bound on r_i $(i \geq 4)$ based on r_4. ♢

Exercise 9.5: Let $\mu > 1$ be integer and $0 < \rho < 1$ a real number. Think of μ as arbitrary but ρ as fixed. We want the integer expression $E(\mu)$ such that

$$E(\mu) > \mu\rho \geq E(\mu) - 1.$$

If ρ is irrational, then $E(\mu) = \lceil \mu\rho \rceil$ is the desired expression. What if $\rho = p/q$ is a rational? ♢

Exercise 9.6: It is well known that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$ when n is integer. When n is non-integer, show that

$$n - 1 < \lfloor n/2 \rfloor + \lceil n/2 \rceil < n + 1$$

and this is the sharpest possible bound. ♢

Exercise 9.7: In a certain household, the laundry is sorted into two bins with the labels *Whites and Underclothing Only*, and *All Others*. What domestic catastrophe ensues when a family member reads one sign as *White and Underclothing Only*? Which items are misplaced by this member? HINT: "White" is a predicate and "Whites" is a noun. ♢

Exercise 9.8: Consider the following sentence:

$$(\forall x \in \mathbb{Z})(\exists y \in \mathbb{R})(\exists z \in \mathbb{R}) \left[(x > 0) \Rightarrow ((y < x < y^{-1}) \wedge (z < x < z^2) \wedge (y < z)) \right] \tag{34}$$

Note that the range of variable x is $\mathbb{Z}$, not $\mathbb{R}$. This is called a **universal sentence** because the leading quantifier is the universal quantifier $(\forall)$. Similarly, we have **existential sentence**.

(i) Negate the sentence (34), and then apply De Morgan's law to rewrite the result as an existential sentence.

(ii) Give a counter example to (34).

(iii) By changing the clause "$(x > 0)$", make the sentence true. Indicate why it would be true.

♢

Exercise 9.9: Let $f(n) = (\log n)^{\log n}$. Suppose you want to prove

$$f(n) \neq \mathcal{O}(f(n/2)).$$

(a) Using de Morgan's law, show that this amounts to saying that for all $C > 0, n_0$ there exists n such that

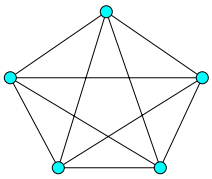
$$(n \geq n_0) \wedge f(n) > Cf(n/2).$$

(b) Complete the proof by finding a suitable n for any given C, n_0. ♢

Exercise 9.10: The following assertion is valid: *a planar graph on n vertices has at most $3n-6$ edges.* Let us restate it as follows:

$$(G \text{ is a planar graph and has } n \text{ vertices}) \Rightarrow (G \text{ has} \leq 3n-6 \text{ edges}).$$

(i) State the contra-positive of this statement.
(ii) The complete graph on 5 vertices is denoted by K_5. Prove that K_5 is not planar by showing the the contra-positive statement. ♢



Complete graph K_5

Exercise 9.11: The road map of a certain state has 314, 159 road segments, where each road segment connects a pair of distinct junctions. Assume that every road segment connect a different pair of junctions. What is the least number of junctions in that state? HINT: use the theorem in the previous exercise about planar graph. ♢

Exercise 9.12: Let $P : D \to \{0, 1\}$ be a partial predicate over some domain D. When we do quantification, $\forall x$ and $\exists x$ it is assumed that x range over D. Show that the following equivalences (called "de Morgan's laws for quantifiers") hold:
(a) $\neg(\forall x)P(x)$ is equivalent to $(\exists x)\neg P(x)$
(b) $\neg(\exists x)P(x)$ is equivalent to $(\forall x)\neg P(x)$ ♢

Exercise 9.13: We say that a tree is **ordered** if the children of each node are totally ordered (so we can speak of the first child, the last child, etc). What is wrong with defining a **binary tree** as a rooted, ordered tree of degree 2? ♢

Exercise 9.14: Prove these basic facts about binary trees: assume $n \geq 1$.
(a) A full binary tree on n leaves has $n-1$ internal nodes.
(b) Show that every binary tree on n nodes has height at least $\lceil \lg(1+n) \rceil - 1$. HINT: define $M(h)$ to be the maximum number of nodes in a binary tree of height h.
(c) Show that the bound in (b) is tight for each n.
(d) Show that a binary tree on $n \geq 1$ leaves has height at least $\lceil \lg n \rceil$. HINT: use a modified version of $M(h)$.
(e) Show that the bound in (d) is tight for each n. ♢

Exercise 9.15: (Erdös-Rado) Show that in any 2-coloring of the edges of the complete graph K_n, there is a monochromatic spanning tree of K_n. HINT: use induction. ♢

Exercise 9.16: Let T be a binary tree on n nodes.
(a) What is the minimum possible number of leaves in T?
(b) Show by strong induction on the structure of T that T has at most $\lfloor \frac{n+1}{2} \rfloor$ leaves. This is an exercise in case analysis, so proceed as follows: first let n be odd (say, $n = 2N+1$) and assume T has $k = 2K+1$ children in the left subtree. There are 3 other cases.
(c) Give an alternative proof of part (b): show the result for n by a weaker induction on $n-1$ and $n-2$.
(d) Show that the bound in part (b) is the best possible by describing a T with $\lfloor \frac{n+1}{2} \rfloor$ leaves. HINT: first show it when $n = 2^t - 1$. Alternatively, consider binary heaps. ♢

Exercise 9.17:
(a) A binary tree with a key associated to each node is a binary search tree iff the in-order

listing of these keys is in non-decreasing order.
(b) Given *both* the post-order and in-order listing of the nodes of a binary tree, we can reconstruct the tree. ♢

End Exercises

§10 APPENDIX B: The RAM Model

Comparison tree models are somewhat special because they are non-uniform. We now present a uniform computational model called the **Random Access Memory model** (RAM model, for short).

¶A.13 Universal Computational Models.
A well-known universal model is the Turing machine. A model is **universal** if every solvable problem can be solved by programs in the universal model. Universality seems like a tall order; indeed, universality is not established by a theorem but by agreement among experts! This agreement is called **Church's Thesis**. But once we agree to call one model "universal", then the universality of others can[20] be established by theorems. To achieve the power of universality, the data structures in the universal model must be general enough to encode any other data structure or data objects.

¶A.14 Register Model.
The RAM model can be viewed as an abstract form of a simple assembly language. In turn, the RAM model is a generalization of an even simpler model called the **Register model**. So we shall begin by describing such "register machines".

(a) Data objects. We assume infinitely many **(storage) registers**, each indexed by an integer $0, \pm 1, \pm 2, \pm 3, \ldots$. Register 0 is special and is known as the **accumulator**. Each register can store an integer. The integer can be arbitrarily large.

this is not a 64-bit or even 128-bit machine...

(b) Primitive Operations. In the simplest form, each operation has 2 fields:

$$\langle OPERATOR \rangle \langle ARG \rangle$$

There is one argument $\langle ARG \rangle$ whose nature is determined by the $\langle OPERATOR \rangle$: $\langle ARG \rangle$ is either an integer (denoted n below) or a label (denoted ℓ below). But in general, an operation can have up to 4 fields:

$$[\langle LABEL \rangle :] \langle OPERATOR \rangle \langle ARG \rangle [\langle COMMENTS \rangle]$$

where $\langle LABEL \rangle$ and $\langle COMMENTS \rangle$ are arbitrary non-empty alphabetic strings – the square brackets indicate that these are optional fields. The contents of register n is a number, denoted $c(n)$. Thus the **contents function** has the form $c : \mathbb{Z} \to \mathbb{Z}$. The operators, their arguments and actions are specified in Table 1.

Most of these operations have the obvious meaning. For the DIV operation, we assume that the integer quotient is put into $c(0)$ and any remainder is discarded.

(c) Semantics. A **register program** P is any finite sequence of such primitive operations. There is also a **label** function λ which, for any label ℓ, returns the index $\lambda(\ell)$ of the instruction

[20] Technically, by "encoding and simulation".

in this sequence P. Now we can define a **computation** in which at each instance, we have a **program counter** whose value is the index of the (current) instruction in P being executed. When we execute the current instruction, this results in a transformation of the contents function. This transformation is specified by the last column of Table 1. For instance, "GET 4" will put update the value of $c(0)$ to be the value $c(4)$, but no other register contents are changed. Subsequently, transformations are defined by successive instructions of the program (this means that the program counter is simply incremented). The exception is when there is a successful jump to some label ℓ, in which case the program counter is updated to $\lambda(\ell)$. The computation halts on encountering the HALT operation, or when there is no "next" instruction, or upon jumping to some non-existent label.

Operator	Argument	Semantics
GET	n	$c(0) \leftarrow c(n)$.
PUT	n	$c(n) \leftarrow c(0)$.
ZERO	n	$c(n) \leftarrow 0$.
INC	n	$c(n) \leftarrow c(n) + 1$.
ADD	n	$c(0) \leftarrow c(0) + c(n)$.
SUB	n	$c(0) \leftarrow c(0) - c(n)$.
MUL	n	$c(0) \leftarrow c(0) \times c(n)$.
DIV	n	$c(0) \leftarrow c(0) \div c(n)$, error if $c(n) = 0$.
JUMP	ℓ	Go to label ℓ.
JPOS	ℓ	If $c(0) > 0$ then go to ℓ.
JNEG	ℓ	If $c(0) < 0$ then go to ℓ.
HALT		The computation halts.

Table 1: Instruction Set for Register Machines

(d) Input/Output conventions. We assume that a finite number of registers is initialized with an encoding of the input, while the rest of the registers are initially zero. The convention for the output is some simple function of the final contents function. E.g., the output may be defined to be $c(0)$. As an exercise, the reader may write a RAM program to compute the maximum of three numbers. Use the convention that the input numbers are in registers $1, 2, 3$ and the maximum value must be output in register 0).

¶A.15 Random Access Feature.

It is a very small step to turn the above Register Model into a RAM Model. First notice that a register program can only access a fixed set of registers. In order to allow a program to access an arbitrary number of registers, we introduce "indirect addressing", a concept from computer architecture. We allow another form of integer argument, denoted $@n$ where $n \in \mathbb{N}$. This means we are using the value $c(c(n))$ instead of $c(n)$ as the actual argument for the operation. Thus $c(n)$ is interpreted as the address of the actual argument. We call $@n$ an "indirect argument". For instance, "GET @4" results in the assignment $c(0) \leftarrow c(c(4))$. If register 4 contains 256, and register 256 contains -1, this means $c(0)$ is assigned the value -1. Similarly, "PUT @4" will place the value $c(0)$ into register 256. Thus, a **RAM program** is basically a register program in which we allow indirect addressing.

¶A.16 Extensions.

By design, the instruction set of our RAM is parsimonious and primitive. There are many possible extensions where we enrich the instruction set; these makes programming convenient, but do not extend the power of the model. For instance, we can allow another kind of integer argument denoted "$= n$". This means that the value n itself is being used – we never have to access the contents function c. This is called a **literal argument**. Literal arguments are useful for GET and the arithmetic instructions, but meaningless for PUT instruction.

The above model may also be called the **Integer RAM model**. This can be generalized to the **Real RAM model** where where the registers can store an arbitrary real number, and possibly augmenting the primitive operations with other real functions (such as computing square roots). When we use a register n for indirect addressing ($@n$), we need to have some convention for handling the case where its contents $c(n)$ is not an integer. However, we must be aware that the Real RAM is a highly non-trivial extension; it is a useful model but it is not realistic by any stretch of imagination. One reason is that the set of real numbers are uncountable and we have no hope of representing all real numbers in a finitistic scheme.

We can also augment the model with new primitive operations. For instance, to write programs in the Tape Model (see ¶7), we just have to add operations corresponding to the `READ`, `WRITE`, `RESET` in (4), and the `EOT` test.

¶A.17 Higher Level Languages and Universality.

In practice, we may write our program using a more abstract language or a "higher level" language. Thus, we may use well-known constructs such as for-loops and if-then-else, and even allow recursion. Nevertheless, such extensions of the model can be translated into a standard RAM program. In this sense, the RAM model is universal in the sense that there is no computational model that is more powerful. In complexity theory, this claim about "universality" is called **the Church-Turing Thesis**.

EXERCISES

Exercise 10.18: Argue that the (Integer) RAM model is equivalent to the Turing model.
HINT: This amounts to showing that for any Turing machine M (resp., any RAM program P), there is a RAM Program P' (resp., a Turing machine M') such that P' simulates M (resp., M' simulates P). ♢

Exercise 10.19: Recall the Merge Algorithm described in ¶9. Please convert it into a RAM program. In other words, you must use the instruction set in Table 1. ♢

Exercise 10.20: Write RAM algorithms for the following problems:
(a) Sort a sequence of input numbers.
(b) Compute the GCD of two integers.
(c) Solve the ranking problem in ¶4. You need two algorithms, for preprocessing and for answering queries.
(d) Solve the dynamic ranking problem in ¶5. You need four algorithms here: to initialize an empty data structure, to insert, to delete, and to answer rank queries.
Be sure to state your input/output conventions. Also, describe any data structures you use. ♢

Exercise 10.21: Reduce the number of primitive operators (listed in table above) for our RAM model to a minimum. In particular, show that we only need the following:

$$GET, PUT, ZERO, INC, JPOS, HALT$$

Show that a RAM model with this set of instructions can simulate our original RAM model. ♢

Exercise 10.22: Show how to implement higher level language constructs such as *for-loops, if-then-else, case-statements* in our RAM model. ♢

END EXERCISES

References

[1] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[2] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

[3] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Boston, 1972.

[4] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Boston, 2nd edition edition, 1975.