Lecture I
OUTLINE OF ALGORITHMICS

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

Algorithmics concern small problems

READING GUIDE: This chapter is mostly informal and depends on some prior understanding of algorithms. The rest of this book has no dependency on this chapter, save the definitions in §8 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.

§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.

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 of discipline X. Thus, we have sorting techniques, graph searching, string algorithms, string algorithms, dynamic programming, numerical PDE, etc, that cuts across individual disciplines. A good way to represent this data is to use a matrix:

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

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

Computer Science is row-oriented

So each of the computational X's represents a column in this matrix, and each computational technique represents a row. Each checkmark indicates that a particular computational technique is used in a particular discipline X. Individual scientific disciplines take a column-oriented view, but Computer Science takes the row-oriented view. These row labels can be classified into four basic themes:

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

(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. Or, an algorithmic technique may entail certain analysis methods (e.g., divide-and-conquer algorithms require recurrence solving). Complexity theory 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, say, more like combinatorics (which is eclectic) than group theory (which starts out from a unified framework).

§2. What are Computational Problems?

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

¶1. (A) Input-output problems. Here is the simplest formulation: A **computational problem** is a precise specification of input and output formats, and for each input instance I, a description of the set of possible output instances $O = \mathcal{O}(I)$.

Standard I/O problems

The word "formats" emphasizes the fact the input and output representation is part and parcel of the problem. 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.

▌**Example:** (A1) Sorting Problem: 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 $(1, 2, 2, 5, 7)$.

¶2. (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

▌**Example:** (B1) Ranking Problem: 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 is S is the number of items in S that are smaller than or equal to q. A standard solution to this problem is the *binary search tree* data structure $D(S)$ and the binary search algorithm on $D(S)$.

▌**Example:** (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 $D_1(S)$ is less efficient than $D_2(S)$. But our general attitude 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.

Preprocessing problems can be seen as 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 anticipate all possible extensions of the partial inputs.

¶3. (C) Dynamization and Online problems. Now assume the input S is a set, or more generally some kind of aggregate object. 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, answer 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**. 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.

Example: (C1) Dynamic Ranking Problem: Any preprocessing problem can be systematically converted into a set maintenance problem. For instance, the ranking problem turns into the **dynamic ranking problem** in which we dynamically maintain the set S subject to intermittent rank queries. The data structures in solutions to this problem are usually called **dynamic search trees**.

Example: (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.

¶4. (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 languages 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.

¶5. Models for Sorting. To illustrate computational models, we consider the problem of sorting. The sorting problem has been extensively studied since the beginning of Computer Science (from the 1950's). It turns out that there are several computational models underlying this simple problem, each giving rise to distinct computational issues. We briefly describe just three of them: the **comparison-tree model**, the **comparator circuit model**, and the **tape model**. In each models, the data objects are elements from a linear order.

3 sorting models

The first model, comparison-trees, 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 represent flow of control and are more generally called **decision trees**. Figure 1(a) illustrates a comparison-tree on inputs x, y, z. The output of the decision tree is specified at each leaf. For instance, if the tree is used for sorting, we would want to write the sorted order of the input elements in each leaf. If the tree is used to find the maximum element of the input set, then each leaf would specify the maximum element.

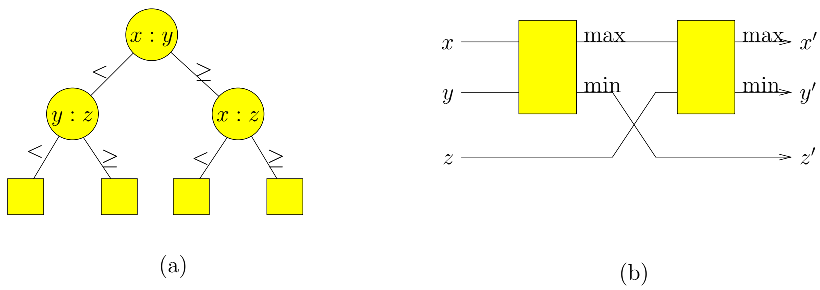

Figure 1: (a) A comparison-tree and (b) a comparator circuit

In the comparator circuit model, we also have one primitive operation which takes two input elements x, y and returns two outputs: one output is $\max\{x, y\}$, the other $\min\{x, y\}$. These are composed into **circuits** which are directed acyclic graphs with n input nodes (in-degree 0) and n output nodes (out-degree 0) and some number of comparator nodes (in-degree and out-degree 2). In contrast to tree programs, the edges (called **wires**) in such circuits represent actual data movement. Figure 1(b) shows a comparator circuit on inputs x, y, z. Depending on the problem, the output of the comparator circuit may be the set of all output lines (x', y', z' in Figure 1(b)) or perhaps some subset of these lines.

A third model for sorting is the tape model. A tape is a storage medium which allows slow, sequential access to its data. We can use several tapes and limited amount of main memory, and the goal is to minimize the number of passes over the entire data. We will not elaborate on this model, but [3] is a good reference. Tape storage was the main means of mass storage in the early days of computing. Curiously, some variant of this model (the "streaming data model") is becoming important again because of the vast amounts of data to be process in our web-age.

¶6. Algorithms versus programs. To use a computational model to solve a given problem, we must make sure there is a match between the data objects in the problem specification and the data objects handled by the computational model. If not, we must specify some suitable encoding of the former objects by the latter. Similarly, the input and output formats of the problem must be represented in some way. After making explicit such encoding conventions, we may call A an **algorithm for** P if, if the program A indeed computes a correct output for every legal input of P. Thus the term algorithm is a semantical concept, signifying a program in its relation to some problem. In contrast, programs may be viewed as purely syntactic objects. E.g., the programs in figure 1(a,b) are both algorithms to compute the maximum of x, y, z. But what is the output convention for these two algorithms?

¶7. Uniform versus Non-uniform Computational Models. While problems generally admit inputs of arbitrarily large sizes (see discussion of size below), some computational models define programs that admit inputs of a fixed size only. This is true of the decision tree and circuit models of computation. In order to solve problems of infinite sizes, we must take a sequence of programs $P = (P_1, P_2, P_3, \ldots)$ where P_i admits inputs of size i. We call such a program P a **non-uniform program** since we have no *á priori* connections between the different P_i's. For this reason, we call the models whose programs admit only finite size inputs **non-uniform models**. The next section will introduce a **uniform model** called the RAM Model. Pointer machines (see Chapter 6) and Turing machines are other examples of uniform models. The relationship between complexity in uniform models and in non-uniform models is studied in complexity theory.

¶8. Problem of Merging Two Sorted Lists. Let us illustrate the difference between uniform and non-uniform algorithms. 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. Here is a algorithm for this problem, written in a generic conventional programming language:

```
MERGE ALGORITHM
Input: (x_1,...,x_m) and (y_1,...,y_n), sorted in non-decreasing order.
Output: The merger of these two lists, in non-decreasing order.
    ▷ Initialize
        i ← 1, j ← 1.
    ▷ Loop
        while (i ≤ m and j ≤ n)
            if x_i ≤ y_j
                Output(x_i) and i ← i + 1.
            else
                Output(y_j) and j ← j + 1.
    ▷ Terminate
        if i > m    ◁ The x's are exhausted
            Output the rest of the y's.
        else  ◁ The y's are exhausted
            Output the rest of the x's.
```

The student should note the conventions used in this program, which will be used throughout. First, we use indentation for program blocks. Second, we use two kinds of comments: (▷ *forward comments*) and (◁ *backward comments*).

Program conventions!

This Merge Algorithm is a uniform algorithm for merging two lists. For each m, n, this algorithm can be "unwounded" into a comparison-tree $T_{m,n}$ for merging two sorted lists of sizes m and n (Exercise). Hence family $\{T_{m,n} : m, n \in \mathbb{N}\}$ is a non-uniform algorithm for merging two lists.

¶9. Program Correctness. This has to do with the relationship between an program and a computational problem. *A program that is correct relative to a problem is, by definition, an algorithm for that problem.* It is usual to divide correctness into two parts: partial correctness and halting. Partial correctness says that the algorithm gives the correct output provided it halts. In some algorithms, correctness may be trivial but this is not always true.

Exercises

Exercise 3.1: What problems do the programs in Figure 1(a) and (b) solve, respectively? You have some leeway in giving them suitable interpretations. ♢

Exercise 3.2: (a) Extend the program in Figure 1(a) so that it sorts three input elements $\{x, y, z\}$. (b) In general, define what it means to say that a comparison-tree program sorts a set $\{x_1, \ldots, x_n\}$ of elements. ♢

Exercise 3.3: Design a tree program to merge two sorted lists (x, y, z) and (a, b, c, d). The height of your tree should be 5 (the optimum). ♢

Exercise 3.4: Draw the tree program corresponding to unwinding the Marge Algorithm on input (x_1, x_2) and (y_1, y_2, y_3, y_4). This is called $T_{2,4}$ in the text. ♢

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. In our discussions, we focus on only one resource at a time, usually time (occasionally space). So we avoid issues of trade-offs between two resources.

Next, for each primitive operation executing on a particular data, we need to know how much of the resource is consumed. For instance, in `Java`, we could define each execution of the addition operation on two numbers a, b to use time $\log(|a| + |b|)$. But it would be simpler to say that this 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 data.*

How is the running time for sorting 1000 elements related to the running time for sorting 10 elements? The answer lies in viewing running time as a function of the number of input elements, the "input size". In general, problems usually have a natural notion of "input size" and this is the basis for understanding the complexity of algorithms.

So we want a notion of **size** on the input domain, and measure resource usage as a function of input size. The size $size(I)$ of an input instance I is a positive integer. We make a general assumption about the size function: *there are inputs of arbitrarily large size.*

For our running example of the sorting problem, it may seem natural to define the size of an input $(a_1, \ldots, a_n)$ to be n. But actually, this is only natural because we usually use computational models that compares a pair of numbers in unit time. For instance, if we must encode the input as binary strings (as in the Turing machine model), then input size is better taken to be $\sum_{i=1}^{n}(1 + \log(1 + |a_i|))$.

Suppose A is an algorithm for our problem P. For any input instance I, let $T_A(I)$ be the total amount of time used by A on input I. Naturally, $T_A(I) = \infty$ if A does not halt on I. Then we define the **worst case running time** of A to be the function $T_A(n)$ where

$$T_A(n) := \max\{T_A(I) : size(I) = n\}$$

Using "max" here is only one way to "aggregate" the set of numbers $\{T_A(I) : size(I) \le n\}$. Another possibility is to take the average. In general, we may apply some function G,

$$T_A(n) = G(\{T_A(I) : size(I) \le n\})$$

For instance, if G is the average function and we get **average time complexity**.

To summarize: a **complexity model** is a specification of
(a) the computational resource,
(b) the input size function,
(c) the unit of resource, and
(d) the method G of aggregating.
Once the complexity model is fixed, we can associate to each algorithm A a **complexity function** T_A.

▌**Example:** (T1) Consider the Comparison Tree Model for sorting. Let $T(n)$ be the worst case number of comparisons needed to sort n elements. Any tree program to sort n elements must have at least $n!$ leaves, since we need at least one leaf for each possible sorting outcome. Since a binary tree with $n!$ leaves has height at least $\lceil \lg(n!) \rceil$.

Lemma 1. *Every tree program for sorting n elements has height at least $\lceil \lg(n!) \rceil$, i.e., $T(n) \ge \lceil \lg(n!) \rceil$.*

This lower bound is called the **Information Theoretic Lower Bound** for sorting.

▌**Example:** (T2) In our RAM model (real or integer version), let the computational resource be time, where each primitive operation takes unit time. The input size function is the number of registers used for encoding the input. The aggregation method is the worst case (for any fixed input size). This is called the **unit time** complexity model.

¶10. Complexity of Merging. Define $M(m, n)$ to be the minimum height of any comparison tree for merging two sorted lists of sizes m and n, respectively. We can prove the following bounds

$$M(m, n) \le m + n - 1$$

and

$$M(m, n) \ge 2 \min\{m, n\} - \delta(m, n)$$

where $\delta(m, n) = 1$ if $m = n$ and $\delta(m, n) = 0$ otherwise. The upper bound comes from the algorithm for merging described in §3: each comparison results in at least one output. But the last element can be output without any comparison. Hence we never make more comparisons than $m + n - 1$. The lower bound comes from the following example: assume the input is $x_1 < x_2 < \cdots < x_m$ and $y_1 < \cdots < y_n$ where $m \ge n$. We assume that

Some interesting stuff at last!

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

Note that each y_i must be compared to x_i and x_{i+1} (for $i = 1, \ldots, n-1$). Moreover, y_n must be compared to x_n, and in case $\delta(m, n) = 0$, y_n must also be compared to x_{n+1}. This proves

$M(m, n) \geq 2n - \delta(m, n)$, where $n = \min\{m, n\}$. This method of proving lower bounds is called the **adversary argument**.

A corollary of the above upper and lower bounds are 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).

Now consider the other extreme where the two input lists are as disparate in lengths as possible: $M(m, 1)$. In this case, the information theoretic bound says that $M(m, 1) \geq \lceil \lg(m + 1) \rceil$ (why?). Also, by binary search, this lower bound is tight. Hence we now know another exact value:

$$M(m, 1) = \lceil \lg(m + 1) \rceil .$$

A non-trivial result from Hwang and Lin says

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

More generally, the **information-theoretic bound** says

$$M(m, n) \geq \lg \binom{m+n}{m}$$

since there are $\binom{m+n}{n}$ ways of merging the two sorted lists. To see this, imagine that we already have the sorted list of $m + n$ elements: but which of these elements come from the list of size m? There are $\binom{m+n}{m}$ ways of choosing these elements.

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.

¶11. Other Complexity Measures. There are complexity models. For instance, in computational geometry, it is 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**.

Remarks:
1. 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.
2. The comparison tree complexity model ignores all the other computational costs except comparisons. In most situations this is well-justified. But it is possible[2] to create conjure up ridiculous

[2] My colleague, Professor Robert Dewar suggests 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.

algorithms which minimize the comparison cost, at an exhorbitant cost in other operations.
3. 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? Give a lower bound in the comparison tree model. Note: to do the computation by hand, it is handy to know that $10! = 3,628,800$ and $2^{20} = 1,048,576$.

SOLUTION: The information theoretic lower bound is $T(n) \geq \lceil \lg(10!) \rceil = 22$.
Comments:

◇

Exercise 4.2: How good is the information theoretic lower bound for sorting 3 elements a sharp bound? In other words, can you find upper bounds that matches the information-theoretic lower bound? Repeat this exercise for 4 and 5 elements. ◇

Exercise 4.3: The following is a variant of the previous exercise. Is it always possible to sort n elements using a comparison tree with $n!$ leaves? Check this out for $n = 3, 4, 5$. ◇

Exercise 4.4: (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 4.5: 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 4.6: Justify the claim that $M(m, 1) = \lceil \lg(m + 1) \rceil$. ◇

Exercise 4.7: Using direct arguments, give your best upper and lower bounds for $M(2, 10)$. ◇

SOLUTION: $M(2, 10) \geq 7$: The information-theoretic lower bound of $\lceil \lg \binom{12}{2} \rceil = \lceil \lg 66 \rceil = 7$. Also, $M(2, 10) \leq 8$: we can use binary search to insert each element in the 2-element list into the longer list. This takes $\leq \lceil \lg 11 \rceil + \lceil \lg 12 \rceil = 4 + 4 = 8$ comparisons.
Comments:

Exercise 4.8: Prove that $M(m, m+i) = 2m + i - 1$ for $i = 2, 3, 4$ for $m \geq 6$. ♢

Exercise 4.9: 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 4.10: Open problem: determine $M(m, 3)$ and $M(m, m+5)$ for all m. ♢

Exercise 4.11: With respect to the comparator circuit and tree program models in §3, describe suitable complexity models for each. ♢

End Exercises

§5. Algorithmic Techniques: How to design algorithms

Now that we have some criteria to judge algorithms, we begin to design algorithms. 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 divide-and-conquer: Suppose you want to sort an array A of n elements. Assume n is a power of 2. Here is the Merge Sort algorithm on input A:

1. (Basis) If n is 1 simply return the array A.
2. (Divide) Divide the elements of A into two subarrays B and C of size $n/2$ each.
3. (Recurse) Recursively, call the Merge Sort algorithm on B. Do the same for C.
4. (Conquer) Merge the sorted arrays B and C and put the result back into array A.

There is only one non-trivial step, the merging of two sorted arrays. We leave this as an exercise.

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. The following are some 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[3] 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) Give a pseudo-code description of $Merge(B, C, A)$ which, given two sorted arrays B and C of size n each, returns their merged (hence sorted) result into the array A of size $2n$.
(b) Why did we assume n is a power of 2 in the description of merge sort? How can we justify this assumption in theoretical analysis? How can we handle this assumption in practice? ♢

Exercise 5.2: 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. ♢

End Exercises

§6. Analysis: How to estimate complexity

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 depends to a large extent on the algorithmic paradigm or data structure used by A. We give two examples.

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

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 $T(1) = 1$. Here $T(n)$ is the (worst case) number of comparisons needed by our algorithm to sort n elements. The solution is $T(n) = \Theta(n \log n)$. In the next chapter, we study the solutions of such equations.

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 still logarithmic cost in the **amortized sense**. 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 to something small (say logarithmic). This technique prevents an adversary from repeated querying of a "deep leaf". Unfortunately, this is not enough because the very first query into a "deep leaf" has to be amortized as well (since there may be no subsequent queries). To anticipate this amortization cost, we "pre-charge" the requests (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.

You may forget the rest of this chapter, but not this part!

We started with a problem, selected a computational model and an associated complexity model, designed an algorithm and managed to analyze its complexity. Looking back at this process, we are certain to find arbitrariness in our choices. For instance, would a simple change in the set of primitive operations 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.

¶12. What is a complexity function? In this book, we call a partial real function

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

a **complexity function** (or simply, "function"). We use complexity functions to quantify the complexity of our algorithms. Why do we consider *partial* 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 $T_A(n)$ would be total. So why do we think of $\mathbb{R}$ as the domain of $T_A(n)$? Again, 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 becomes 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 $n \le 0$. If $f(n)$ is not defined at n, we write $f(n) =\uparrow$, otherwise $f(n) =\downarrow$. Since complexity functions are partial, we have to be careful about operations such as functional composition.

¶13. 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 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.

¶14. 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". Two real functions f, g,are **polynomially equivalent** in this sense if for some $c > 0$, $f(n) \le cg(n)^c$ and $g(n) \le cf(n)^c$ 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:

- Step 1: We are interested in the eventual behavior of functions (e.g., if $T(n) = 2^n$ for $n \le 1000$ 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.

¶15. Eventuality. This is Step 1 in our search for invariance. Given two functions, we say "$f \le g$ **eventually**", written

$$f \le g \text{ (ev.)}, \tag{1}$$

if $f(x) \le g(x)$ holds for all x large enough. More precisely, this means there is some x_0 such that the following statement is true:

$$(\forall x)[x \ge x_0 \Rightarrow f(x) \le g(x)]. \tag{2}$$

By not caring about the behaviour of complexity function over some initial values, our complexity bounds becomes robust against the following table-lookup trick. Given any algorithm, it is conceivable that for any finite set of inputs, the algorithm store their answers in a table. This modified algorithm only has to do a table-lookup to provide answers for these cases, and otherwise it operates as before. But this table-lookup algorithm has the same "eventual" complexity as the original algorithm.

REMARK: We must be careful with (2): *what does "$f(x) \le g(x)$" mean when either $f(x)$ or $g(x)$ may be undefined?* The answer depends on the quantifier that bounds (i.e., that controls) x, whether x is bounded by an existential quantifier or by a universal quantifier. If a universal quantifier (as in (1)), we declare the predicate "$f(x) \le g(x)$" to be true if either $f(x)$ or $g(x)$ is undefined. If an existential quantifier, we declare the predicate "$f(x) \le g(x)$" to be false if either $f(x)$ or $g(x)$ is undefined. So (2) can be expanded into:

$$(\forall x)[(x \ge x_0 \wedge \ f(x) =\downarrow \wedge\ g(x) =\downarrow). \Rightarrow .(f(x) \le g(x))].$$

We generalize this treatment of quantification: by a **partial predicate** on real numbers, we mean a partial function $P : \mathbb{R} \to \{0, 1\}$. In the previous example, $P(x)$ is just "$f(x) \le g(x)$". The universally sentence "$(\forall x)[P(x)]$" should be interpreted as saying "for all $x \in \mathbb{R}$, if $P(x)$ is defined then $P(x)$ is true". Similarly, the existentially sentence "$(\exists x)[P(x)]$" says "there exists some $x \in \mathbb{R}$ such that $P(x)$ is defined and $P(x)$ is true".
Note that the sentence "$P(x)$ holds eventually" has the form "$(\exists y)(\forall x)[R(x, y)]$" where $R(x, y) \equiv (x \ge y \Rightarrow P(x))$. More generally, let $R(x, y)$ be a partial predicate. Let us say that x_0 is "eligible in the first argument" if there exists some y_0 such that $R(x_0, y)$ is defined. We can similarly define eligibility in the second argument. Then "$(\forall x)(\exists y)[R(x, y)]$" means that "for all x eligible in the first argument, there exists y such that $R(x, y)$ is defined and is true". In particular, if there are no x eligible in the first argument, the sentence is true. On the other hand, "$(\exists x)(\forall y)[R(x, y)]$" means that "there is an eligible x in the first argument, for all y, if $R(x, y)$ is defined then $R(x, y)$ is true". In this case, there is at least one eligible x in the first argument. This treatment extends easily to any partial predicates on any number of arguments and quantifiers.

To show the role of the x variable, we may also write (1) as

$$f(x) \le g(x) \text{ (ev. } x).$$

Clearly, this is a transitive relation.

The "eventually" terminology is quite general: if a predicate $R(x)$ is parametrized by x in some real domain $D \subseteq \mathbb{R}$, and $R(x)$ holds for all $x \in D$ larger than some x_0, then we say $R(x)$ **holds eventually** (abbreviated, ev.). We can also extend this to predicates $R(x, y, z)$ on several variables. A related notion is this: if $R(x)$ holds for infinitely many values of $x \in D$, we say $R(x)$ **holds infinitely often** (abbreviated, i.o.).

If $g \le f$ (ev.) and $f \le g$ (ev.), then clearly

$$g = f \text{ (ev.)}.$$

Thus means $f(x) = g(x)$ for sufficiently large x, whenever both sides are defined. Most natural functions f in complexity satisfies $f \ge 0$ (ev.) and are non-decreasing eventually.

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

$$f \preceq g,$$

if there exists $C > 0$ such that $f \le C \cdot g$ (ev.). Equivalently, $f \preceq g$ is written as $g \succeq f$. This notation naturally suggests the transitivity property: $f \preceq g$ and $g \preceq h$ implies $f \preceq h$. Of course, the reflexivity property holds: $f \preceq f$. If $f \preceq g$ and $g \preceq f$ then we write

$$f \asymp g.$$

Clearly $\asymp$ is an equivalence relation. The equivalence classes of f is called the Θ-**order** of f; more on this below. If $f \preceq g$ but not $g \preceq f$ then we write

$$f \prec g.$$

E.g., $1 + \frac{1}{n} \prec n \prec n^2$.

In short, the triplet of notations $\preceq, \prec, \asymp$ for real functions correspond to the binary relations $\le, <, =$ for real numbers. The basic properties of domination are suggested by this correspondence: since $x \le y$ and $y \le z$ implies $x \le z$, we might expect $f \preceq g$ and $g \preceq h$ to imply $f \preceq h$ (this is true).

Domination provides "implementation platform" robustness in our complexity results: it does not matter whether you implement a given algorithm in a high level program language like `Java` or in assembly. 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 which predicts that the speed of hardware will keep increasing over time (the end is not in sight yet).

¶17. 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

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

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.

The key asymptotic notation to know!
big-Oh is almost the same as domination!

In other words, if $g = \mathcal{O}(f)$ then there is some $C > 0$ and x_0 such that for all $x \ge x_0$, if $g(x) =\downarrow$ and $f(x) =\downarrow$ then $0 \le g(x) \le Cf(x)$.

E.g., The set $\mathcal{O}(1)$ is the set of functions f that is bounded. 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? A partial explanation is that one common use of the equality symbol has a uni-directional flavor where we transform a formula from an unknown form into a known form, separated by an equality symbol. Our one-way equality symbol for $\mathcal{O}$-expressions lends itself to a similar manipulation. For example, the following sequence of one-way equalities

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

may be viewed as a derivation to show f is at most quadratic.

¶18. 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[4] 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

$$E = F$$

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:

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

- $\mathcal{O}(1)$, 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.

¶19. 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,

$$\mathcal{O}_A(n), \quad \mathcal{O}_1(n^2) + \mathcal{O}_2(n \log n). \tag{3}$$

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. How, it 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 "$f(x, y) = \mathcal{O}(g(x, y))$" seems to be clear enough. In practice, this extension seems little needed.

¶20. Related Asymptotic Notations. The above discussion extends in a natural way to several other related notations.

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 $g \succeq f \succeq 0$. Note that $\Omega(f)$ is empty unless it is eventually non-negative. Clearly, big-Omega is just the reverse of the big-Oh relation: g is in $\Omega(f)$ 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 $C > 0$,

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

Thus g is in $o(f)$ implies $g(n)/f(n) \to 0$ as $n \to \infty$. Also, $o(f) \subseteq \mathcal{O}(f)$. A related notation is this: we say

$$f \sim g$$

if $f = g \pm o(g)$ or $f(x) = g(x)[1 \pm o(1)]$.

So $n + \lg n \sim n$.

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

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

Thus g is in $\omega(f)$ implies $g(n)/f(n) \to \infty$ as $n \to \infty$. Clearly $\omega(f) \subseteq \Omega(f)$.

For each of these notations, we again define the $\circ$-expressions ($\circ \in \{\Omega, \Theta, o, \omega\}$), use the one-way inequality 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 $\circ(f)$ the $\circ$-**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 $f = o(g)$.

¶21. Lower Bounds. We can negate the statement $f = \mathcal{O}(g)$ by writing $f \neq \mathcal{O}(g)$. This statement is a way of stating a lower bound on f, since $f = \mathcal{O}(g)$ states an upper bound on f. Thus we have the following three ways to state lower bounds on a complexity function $f(n)$:

- $f(n) = \Omega(g(n))$.
- $f(n) \neq \mathcal{O}(g(n))$.
- $f(n) \neq o(g(n))$.

Each lower bound on f is less stringent than the previous. See Exercise for how these 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 $\varepsilon > 0$.

¶22. Discussion. There is some debate over the best way to define the asymptotic concepts. There is considerable divergence in the literature on the details. Here we note just two alternatives:
1. 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 $\mathcal{O}(f)$. Our definition, on the contrary, allows us to distinguish[5] between $1+\mathcal{O}(1/n)$ and $1-\mathcal{O}(1/n)$. Note that $g = 1-\mathcal{O}(f)$ amounts to $1-Cf \le f \le 1$ (ev.).

Note that when an big-Oh expression appears in negated form as in $-\mathcal{O}(1/n)$, it is really a lower bound
2. 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 $0 \preceq g$ in our original definition. This definition is attractive because of its simplicity. But with this "simplified definition", $\mathcal{O}(f)$ 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 less-than or greater-than symbol: it is very tempting to write "$f(n) \le \mathcal{O}(g)$" instead of "$f(n) = \mathcal{O}(g)$". At best, this is redundant. The problem is that, once this is admitted, one may in the course of a long derivation eventually write "$f(n) \ge \mathcal{O}(E)$" which is not very meaningful. Hence we regard any use of $\le$ or $\ge$ symbols in $\mathcal{O}$-notations as illegitimate (but see below, (3)).

Perhaps most confusion (and abuse) in the literature arises from the variant definitions of the Ω-notation. For instance, one may have only shown a lower bound of the form $g(n) \ne \mathcal{O}(f(n))$ but this is claimed as a $g(n) = \Omega(f(n))$ result. In other words, the expression "$g = \Omega(f)$" is interpreted to mean that there exists (or for all) $C > 0$ such that for infinitely many x, $g(x) \ge Cf(x)$.

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 "parameterized complexity" is gives another example of bivariate complexity functions (one of the size variables controls the "parameters" of the problem).

Exercises

Exercise 7.1: Assume $f(n) \ge 1$ (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 $f(n) \ge 1$ (ev.) is false. ♢

[5] 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.

SOLUTION: a) Let $f(n) \geq 1$ (ev) and assume that $f(n) = n^{\mathcal{O}(1)}$. Therefore there exists a $g \in \mathcal{O}(1)$ such that $g(n) \leq k$ (ev) and $f(n) \leq n^{g(n)} \leq n^k$ (ev). This shows that $f(n) \in \mathcal{O}(n^k)$. In the other direction, suppose $f = O(n^k)$. Then $f \leq Cn^k$ (ev) for some $C > 1$. Thus $f \leq n^{k+\epsilon}$ (ev) for any $\epsilon > 0$ (we This shows $f = n^{O(1)}$.
b) To find a counterexample, we exploit the fact that if $g = \mathcal{O}(1)$ implies $g \geq 0$ (ev). Let us choose $f(n) = 1/2$. Clearly $f = \mathcal{O}(n)$. But $f(n) \neq n^{\mathcal{O}(1)}$ because if $f(n) = n^{g(n)}$ for some function $g(n)$, then clearly $g(n) < 0$ (ev). But this means $g(n) \neq \mathcal{O}(1)$.
Comments:

Exercise 7.2: Prove or disprove: $f = \mathcal{O}(1)^n$ iff $f = 2^{\mathcal{O}(n)}$. ♢

Exercise 7.3: 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 7.4: For basic properties of the logarithm and exponential functions, see the appendix in the next lecture. Show the following (remember that n is the designated variable). In each case, you must explicitly specify the constants n_0, C, etc, implicit in the asymptotic notations.
(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 7.5: 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).
(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 $\ln f = \mathcal{O}(\ln g)$. HINT: careful!
(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) $f(n) = \Theta(f(n/2))$.

SOLUTION: Only two statements are true:

(a) False. $f = n$ and $g = n^2$.

(b) False. Take f any function that is eventually non-zero. Then take $g = -f$

(c) False. Take $f = 1/2$ and any function g such that $g > 1$. But $\log(f) < 0$ and so, by definition of the $\mathcal{O}$-notation, $f \notin \mathcal{O}(\lg g)$).

This solution seems to take advantage of a technical requirement in the definition of $\mathcal{O}$-notation. HERE is another solution which does not exploit this property (and I think is more insightful): Let $f = 2$ and $g(x) = (x+1)/x = 1 + (1/x)$. Then $\ln g > 0$ but $\lg g(n) \to 0$ as $x \to \infty$. [In fact, $\lg g(x) < 1/(2x)$, but you don't need to not know this]. Clearly, $\lg f = 1$ but there is no constant $C > 0$ such that $\lg f = 1 \le C \lg g$.

(d) True. We have $f = \mathcal{O}(g)$ implies there is some $C > 0$ and x_0 such that for all $x > x_0$, $f(x) \le Cg(x)$. Thus, $f(\log(x)) \le Cg(\log(x))$ for all $x \ge e^{x_0}$.

(e) False. Let $f = 2n$ and $g = n$.

(f) True. $f = o(g)$ implies that for all $C > 0$, $0 \le f \le C * g$ (ev). Taking $C = 1$, we obtain that $0 \le 2^f \le 2^g$ or in other words $2^f \in \mathcal{O}(2^g)$.

(g) False. Let $f = 1/n$.

(h) False. $f = 2^n$.

Comments:

♢

Exercise 7.6: Re-solve the previous exercise, assuming that $f, g \ge 2$ (ev.).

SOLUTION: Many statements are now true:

(a) False, as before.

(b) True. $\max(f, g) \le f + g$ (ev), since both functions are ≥ 0 (ev). But $f + g \le 2\max(f, g)$ (ev).

(c) True. Since $f = \mathcal{O}(g)$, we have $f \le Cg$ (ev) for some $C > 1$. So $\lg f \le \lg C + \lg g$. As $g \ge 2$ (ev), so $\lg g \ge 1$ (ev) and so $\lg f \le \lg C + \lg g = (1 + \lg C)\lg g$ (ev).

Alternative proof (based on limits, which I generally avoid): the limit as n goes to infinity of $\ln(f)/\ln(g) = (1/f)/(1/g) = g/f = C$, we see that $\lg(f) = \mathcal{O}(\lg(g))$.

(d) True, as before.

(e) False, as before.

(f) True. as before.

(g) True. We have $1 \le f$ (ev) and hence $f \le f^2$ (ev). Thus $f = \mathcal{O}(f^2)$.

(h) False, as before.

Comments: Some students treat $\max\{f, g\}$ to be either equal to the function f (or g). But this is not true since $\max\{f, g\}$ is a pointwise maximum, which is sometimes attained by f and other times by g.

♢

Exercise 7.7: 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.

SOLUTION:
(i) CLAIM: $f \preceq g$. Choose $C = 1$. Then for all $x \in \mathbb{R}$, we have $f(x) = \sin x \leq 1 = g(x)$. So any choice of x_0 will do.
(ii) Note that $f \preceq g$ fails because f is periodic. Hence we will prove the negation: CLAIM: $g \not\preceq f$. To see this, note that for all $C > 0$ and x_0, there exists $x > x_0$ such that $f(x) = 0$. Hence $g(x) \leq Cf(x)$ does not hold.

Comments:

♢

Exercise 7.8: This exercise shows three (increasingly strong) notions of lower bounds. Suppose $T_A(n)$ is the running time of an algorithm A.
(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 $T_A(n) = \Omega(f(n))$? ♢

Exercise 7.9: Show some examples where you might want to use "mixed" asymptotic expressions. ♢

Exercise 7.10: 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. Two Dictums of Algorithmics

We discuss two principles in algorithmics. They justify many of our procedures and motivate some of the fundamental questions we ask.

(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 of course several caveats: A consequence of this dictum is that a "new" algorithm is not considered significant unless its asymptotic order is less than previous known algorithms. This attitude could be counter-productive if it is abused. Often, an asymptotically superior algorithm may be inferior when compared to another slower algorithm on all inputs of realistic sizes. For special problems, we might be interested in constant multiplicative factors.

(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 the 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.

§A. APPENDIX: 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 to come back again!

§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 function as follows:

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

Again, to define the special symbol for logarithm to base 2, we will say: let $\lg x := \log_2 x$.

§A.1 Numbers.

Denote the set of natural numbers[6] 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 $\mathbb{C}$. 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 $\lceil -2.3 \rceil = -2$.

§A.2 Sets.

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 $X \uplus Y$. Thus, $X = X_1 \uplus X_2 \uplus \cdots \uplus X_n$ to denote a partition of X into n subsets. 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 a n-set refers to one with cardinality n, and $\binom{X}{n}$ denotes the set of n-subsets of X.

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 $\{a, b, c\}$, and we use set notations for 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 $a \in S$.

§A.3 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. 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 say a total function f is **injective** or $1-1$ if $f(x) = f(y)$ imples $x = y$; it is **surjective** or **onto** if $f(X) = Y$; 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 deserve their own notations: $\lg x$ and $\ln x$ refer to logarithms

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

to base $b = 2$ and base $b = e = 2.718...$, respectively. For any real i, we write $\log^i x$ as short hand for $(\log x)^i$. E.g., $\log^2 x = (\log x)^2$. But if i is a natural number then $\log^{(i)} x$ will 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.4 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: $\neg A, A \vee B, A \wedge B$. Note that $A \Rightarrow B$ is logical implication, and is equivalent to $\neg A \vee B$.

But mathematical facts goes beyond propositional logic. Here is an example[7] of a mathematical assertion $P(x, y)$ where x, y are real variables:

$$P(x, y): \text{ There exists a real } z \text{ such that if } x < y \text{ then } x < z < y. \tag{4}$$

The student should know how to parse such assertions. The assertion $P(x, y)$ happens to be true. This is logically equivalent to

$$(\forall x, y \in \mathbb{R})[P(x, y)]. \tag{5}$$

All mathematical assertions are of this nature. Note that we have passed from propositional logic to quantifier (first order) logic. It is said that mathematical truths are universal: truthhood does not allow exceptions. If an assertion $P(x, y)$ has exceptions, and we can explicitly characterize the exceptions $E(x, y)$: then the new statement $P(x, y) \vee E(x, y)$ constitute a true assertion.

Assertions contain variables: for example, $P(x, y)$ in (4) contains x, y, z. Each variable has an implied or explicit range (x, y, z range over "real numbers"), and each variable is either **quantified** (either by "for all" or "there exists") or **unquantified**. Alternatively, they are either **bounded** or **free**. In our example $P(x, y)$, z is bounded 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 (5) 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 is 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

$$(Q_1)(Q_2)\cdots(Q_n)\,[\ldots predicate \ldots]$$

where Q_i is the ith quantifier part. Such a form, where all the quantifiers appear before the predicate part, is said to be in **prenex form**.

For a course in Algorithmics, there is a natural place to practice quantifier logic: in the asymptotic notations.

§A.5 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 can also

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

arise (Chapter 2 treats induction). Proofs by contradiction usually has a creative element: you need to find an assertion to be contradicted!

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 becomes the following: $\neg((\forall x)P)$ is equivalent to $(\exists x)(\neg P)$; $\neg((\exists x)P)$ is equivalent to $(\forall x)(\neg P)$. A useful place to exercise these rules is to do some proofs involving the asymptotic notation (big-Oh, big-Omega, etc). See Exercise.

§A.6 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[8] $|w|$. 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.7 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[9] to denote edge-hood by the notation $u{-}v$. Of course, in the case of bigraphs, $u{-}v = v{-}u$.

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 presense 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 $W : V \to \mathbb{R}$, 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 $V = V_1 \uplus V_2$ and $E \subseteq V_1 \times V_2$. It is common to write $G = (V_1, V_2, E)$ in this case. Thus, $K_{m,n} = (V_1, V_2, V_1 \times V_2)$ denotes the **complete bipartite graph** where $m = |V_1|$ and $n = |V_2|$.

[8] 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.

[9] 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".

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 $(v_1 - v_2 - \cdots - v_k)$. A path is **closed** if $v_1 = v_k$ and $k > 1$. Two closed paths are equivalent if the sequence of edges they pass through are the same up to cyclic reordering. An equivalence class of such closed paths is called a **cycle**. The length of a cycle is just the length of any of its representative closed paths. For bigraphs, we require cycles to have length at least 3. A graph is **acyclic** if it has no cycles. Sometimes acyclic bigraphs are called **forests**, and acyclic digraph are called **dags** ("directed acyclic graph").

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 $u - v$. Clearly, connectivity and adjacency are symmetric binary relation. 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 further details on graph-related matters.

§A.8 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[10] 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 **descendent** of v if there is a path from v to u. Every node v is a descendent of itself, called the **improper descendent** of v. All other descendents of v are called **proper**. We may speak of the **child** or **grandchild** of any node in the obvious manner. The reverse of the descendent 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 descendents of u. 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 descendents. 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.

[10] 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".

See Chapter 3 for further details on binary search trees.

§A.9 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 is a good reason why we stop short of writing compilable code – first of all, it is programming language- dependent. The half-life of programming languages is short as compared[11] to mathematical language. Second, compilable code is meant for machine consumption, and that gets in the way of human understanding. Here is the quick run-down on pseudo-PL:

pseudo-PL is appropriately amorphous by design

e.g., `static volatile int n`

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.
- 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.
- 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[12] "=" 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.
- Comments in a program are indicated in two ways: ▷ *This is a forward comment* and ◁ *This is a backward comment*. These comments either precede (in case of forward comment) or follows (in case of backward comment) the code that it describes.

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

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

[11]It would be hard to read mathematical writing from 80 years ago, but the chance that a program written 20 years ago can still compile today is close to zero. So perhaps the half-life of mathematical language is perhaps 40 years as compare to a half-life of 10 years for programming languages.

[12]Programmers often use "=" for assignment and "==" for equality test. But our choice preserves the original meaning of "=".

It is important to note that we use indentation to indicate the block structure of our code. This avoids visual clutter.

§A.10 How to answer algorithmic exercises.
In our exercises, whenever we ask you to give an algorithm, it is best to write in pseudo code. Throughout this book, you will see examples of such pseudo code. We suggest you emulate this 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 language like Java*. Actually, there are other issues. Here is a checklist you can use:

Rule 1 *Take advantage of well-known algorithms.* For instance, if you are invoking a sorting routine or a standard graph traversal algorithm, you just[13] need to indicate this. Of course, you may need to set up the arguments correctly before calling these standard routines.

Rule 2 *Reduce all operations to* $\mathcal{O}(1)$ *time operations.* Only do this when Rule 1 does not apply. But sometimes, achieving $\mathcal{O}(1)$ time may depend on a suitable choice of data structures. If so, you should explain this.

Rule 3 *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!

Rule 4 *Use progressive algorithm development.* Even pseudo code may make no sense 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, you may do this in 3 steps: explain basic ideas, give pseudo code, further explain certain details in the pseudo code.

Rule 5 *Explain and initialize all variables and data structures.* All non-trivial algorithms has some data structures (possibly the humble array). Critical variables (counters, coloring schemes) ought to be explained too. You must show how to initialize them.

Rule 6 *The control structure of the algorithm should very clear.* Most of the algorithms you need to design have simple structures – typically a simple loop or a doubly-nested loops. Occasionally we see triply-nested loops (in dynamic programming or matrix multiplication). The nature of each loop should be follow standard programming constructs (for-loop, while-loop, etc). It seems to be an axiom that if a problem can be solved, then it is solvable by clean loop structures.

Rule 7 *Correctness.* This is an implicit require of all algorithms. In computer science, the standard meaning of correct algorithms is split into two distinct requirements: (1) the algorithm halts, and (2) the output is correct when it halts. Even when we do not ask you to explicitly prove correctness, you should check this yourself. There is a very simple rule that you should use – at the beginning of every loop iteration, you should be able to attach a suitable **invariant** (also called **assertion** in standard programming languages). The correctness of algorithms follow easily if the appropriate invariants hold.

Rule 8 *Analysis and Efficiency.* This is viewed as a more advance requirement. But since this is what algorithmics is about, we view it as part and parcel of any algorithm design. You should always be able to give a big-Oh analysis of your algorithm. In most cases, any non-polynomial solution is probably unnecessarily inefficient.

Exercises

[13] In computing, we call this "code reuse" but others call this "not reinventing the wheel".

Exercise A.1: The following is a useful result about iterated floors and ceilings.
(a) Let n, b be positive integers. Let $N_0 := n$ 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. HINT: use the fact that $N_{i+1} \leq N_i/b + (b-1)/b$.
(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. ♢

SOLUTION: See the solution to the next problem for the general result about iterated floors.
Comments:

Exercise A.2: Let x, a, b be positive real numbers. Show that

$$\lfloor x/ab \rfloor \geq \lfloor \lfloor x/a \rfloor /b \rfloor . \tag{6}$$

When is this an equality? ♢

SOLUTION: Note that $x/ab \geq \lfloor x/a \rfloor /b$. Then inequality (6) follows by taking floors on both sides.
We claim that equality holds when x, a, b are integers. Write $\lfloor x/a \rfloor = (x/a) - \delta$ where $0 \leq \delta < 1$, and $\lfloor \lfloor x/a \rfloor /b \rfloor = (x/ab) - (\delta/b) - \delta'$ where $0 \leq \delta' < 1$. When x, a, b are integers, we get the stronger bound that $0 \leq \delta \leq (a-1)/a$ and $0 \leq \delta' \leq (b-1)/b$. Now, notice that

$$\Delta := (\delta/b) + \delta' \leq (a-1)/(ab) + (b-1)/b \leq (ab-1)/(ab).$$

This implies that $(x/ab) - \Delta = \lfloor x/ab \rfloor$.
Comments: REMARK: There is an analogous result for the ceiling function $\lceil x \rceil$.

Exercise A.3: Consider the following sentence:

$$(\forall x \in \mathbb{Z})(\exists y \in \mathbb{R})(\exists z \in \mathbb{R}) \left[(x > 0) \Rightarrow ((y < x < y^{-1}) \wedge (z < x < z^2) \wedge (y < z)) \right] \tag{7}$$

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

(ii) Give a counter example to (7).

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

SOLUTION: This example is easiest to understand if you (visually) order the various numbers according to the requirements of the sentence:

$$0 < y < z < x < z^2 < y^{-1}$$

where the relative order of y^{-1} and z^2 is unimportant (can be reversed if you like).
(i)

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

(ii)
A counter example is $x = 1$. First note that $y < x < y^{-1}$ implies $y > 0$. If $z < x$, then $z < 0$ because otherwise we have $0 \le z < 1$ and so $z^2 \le z$, which contradicts the requirement of the sentence. But if $z < 0$, then we cannot satisfy $y < z$.
(iii) We change $(x > 0)$ to $(x > 1)$. This removes the counter example. Now, for any $x > 1$, we can always choose z so that $(z < x < z^2)$. Now if we choose a positive y sufficiently small, we can also satisfy the remaining clauses of our sentence. Students also came up with an alternative answer: replace $(x > 0)$ by $(x < 0)$. Then the sentence is true since you can choose $y = x - 1$ and $z = x - 2$.
Comments: Common mistakes: (i) Students think that $\neg(a < b < c)$ is the same as $(a \ge b \ge c)$, or that $\neg(p \Rightarrow q)$ is $(\neg p \Rightarrow \neg q)$. (ii) Students claim sentence is false by giving a particular example of y and z.

♢

Exercise A.4: Suppose you want to prove that

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

where $f(n) = (\log n)^{\log n}$.
(a) Using de Morgan's law, show that this amounts to saying that for all $C > 0, n_0$ there exists n such that

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

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

Exercise A.5: The following statement is a fact: *a planar graph on n vertices has at most $3n - 6$ edges*. Let us restate it as follows:

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

(i) State the contra-positive of this statement.
(ii) The complete graph on 5 vertices, denoted by K_5 is shown in Figure 2. Using the contra-positive statement in part (i), prove that K_5 is not planar.

SOLUTION:
(i) Constra-positive:

$$(G \text{ has} > 3n - 6 \text{ edges}) \Rightarrow (G \text{ is not a planar graph or does not have } n \text{ vertices}).$$

(ii) Let $n = 5$. Then K_5 has $> 3n - 6 = 9$ edges. By the contra-positive of (i), we conclude that K_5 is not a planar graph or does not have 5 vertices. Since it does have 5 vertices, it must not be planar.
Comments:

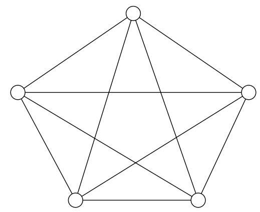

Figure 2: K_5, the complete graph on 5 vertices

◇

Exercise A.6: 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. ◇

SOLUTION: (a) Use structural induction.
BASE CASE: assume n (the number of leaves) is at least 1. Note that a full binary tree, if it has any leaves, must have at least 2 leaves. So $n = 2$ is our base case. The result is clearly true here.
INDUCTIVE CASE: assume $n > 2$. Let $n(T)$ and $i(T)$ denote the number of leaves of T and number of internal nodes of T. Thus, $n = n(T)$. Also, the size of T is $|T| = n(T) + i(T)$.
Let T_L, T_R be the left and right subtrees. Without loss of generality, let $|T_L| \le |T_R|$. If $|T_L| = 1$ then $|T_R| > 1$. By induction, $n(T_R) = i(T_R) + 1$. Then $n(T) = 1 + n(T_R)$ and $i(T) = 1 + i(T_R)$. This proves that

$$\begin{aligned} n(T) &= 1 + n(T_R) && \text{(by definition of } n(T)\text{)} \\ &= 1 + (i(T_R) + 1) && \text{(by induction hypothesis)} \\ &= 1 + i(T) && \text{(by definition of } i(T)\text{)} \end{aligned}$$

In general, we have $|T_L| > 1$, then we have

$$\begin{aligned} n(T) &= n(T_L) + n(T_R) && \text{(by definition of } n(T)\text{)} \\ &= (i(T_L) + 1) + (i(T_R) + 1) && \text{(by induction hypothesis)} \\ &= 1 + i(T) && \text{(since } i(T) = 1 + i(T_L) + i(T_R)\text{)} \end{aligned}$$

(b) Then we have $M(h) = 1 + 2M(h-1)$ and $M(0) = 1$. Thus $M(h) = 2^{h+1} - 1$. Thus we have $n \le M(h)$ and $n + 1 \le 2^{h+1}$ and, by taking logarithm, $h \ge \lg(n+1) - 1$. Since h is an integer, we can take ceiling of $\lg(n + 1)$.
REMARK: this technique of defining $M(h)$ is a good general way to prove lower bound on heights. There is an analogous technique to prove upper bounds on height (see Chapter on AVL trees).
(c) We must show that for every $n \ge 1$, there is a binary tree T_n on n nodes with height $h(n) := \lceil \lg(1 + n) \rceil - 1$. If $n = 1$, $h(1) = 0$, and the result is true. Suppose $n > 1$ and n is even. Then

$$\begin{aligned} h(n) &= \lceil \lg(1 + n) \rceil - 1 \\ &= \lceil \lg((1 + n)/2) \rceil \\ &= \lceil \lg(1 + (n/2)) \rceil && \text{(since } n \text{ even)} \\ &= h(n/2) + 1 \end{aligned}$$

By induction hypothesis, there is a binary tree $T_{n/2}$ on $n/2$ nodes of height $h(n/2)$. Consider the binary tree T_n on n nodes where the left subtree is $T_{n/2}$ and the right subtree has $T_{(n/2)-1}$. The height of T_n is $1 + h(n/2)$, which is equal to $h(n)$. What if n is odd? In this case,

$$\begin{aligned} h(n) &= \lceil \lg(1 + n) \rceil - 1 \\ &= \lceil \lg((1 + n)/2) \rceil \\ &= \lceil \lg(1 + (n-1)/2)) \rceil \\ &= h((n-1)/2) + 1 \end{aligned}$$

Now construct T_n so that its left and right subtrees are both $T_{(n-1)/2}$. Hence T_n has height $1 + h((n-1)/2)$ which is equal to $h(n)$.
(d) We can use the same method as (b), but define a modified function $M'(h)$ to be the maximum number of leaves in a binary tree of height h. Then $M'(h) = 2M'(h-1)$ where $M'(1) = 2$. Then $M'(h) = 2^h$. Hence $n \le M'(h) = 2^h$. Hence $h \ge \lg n$, and again we can take ceiling.
Comments:

Exercise A.7: (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. $\diamondsuit$

SOLUTION: The result is trivial if K_n is monochromatic. Otherwise, take a vertex v that is incident to edges of both colors. Now use induction on K_{n-1} which is obtained by eliminating v. This is a basic result in so-called "Ramsey Theory".
Comments:

Exercise A.8: 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. $\diamondsuit$

SOLUTION:
(a) One. This happens when all but one node has exactly one child.
(d) Fill the nodes of the tree level by level, from left to right.
Comments: REMARK: Students who could not do (b) is basically stumped by not knowing simple "case analysis". Question (c) calls for a construction of a tree with n nodes having exactly $\lfloor \frac{n+1}{2} \rfloor$ leaves.

Exercise A.9:
(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. $\diamondsuit$

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.

[5] C. K. Yap. Introduction to the theory of complexity classes, 1987. Book Manuscript. Preliminary version (on ftp since 1990),
URL `ftp://cs.nyu.edu/pub/local/yap/complexity-bk`.