Lecture I
OUTLINE OF ALGORITHMICS

We assume the student is familiar with computer programming and has a basic course in algorithms and data structures. Problems solved using computers can be roughly classified into problems-in-the-large and problems-in-the-small. The former is usually associated with large software systems such as an airlines reservation system, compilers or text editors. The latter [1] is identified with specific, well-defined problems such as sorting, multiplying two matrices or solving a linear program. The methodology for studying the 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, problems-in-the-small are called **computational problems**, or simply, "problems".

READING GUIDE: The discussions in this chapter are informal and depends on some prior understanding of algorithms. The rest of this book only depends on §8 of this chapter where the important asymptotic notations are introduced.

§1. What is Algorithmics?

Algorithmics is the systematic study of efficient algorithms for computational problems; this includes studying techniques of algorithm design 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 understand them. While algorithmics provide the building blocks for more complex algorithms and large application systems, the construction of large systems usually require additional non-algorithmic techniques (e.g., database theory) which are outside our scope.

One classification of algorithmics is according to its applications in subfields of mathematics and the science: thus we have computational geometry, computational topology, computational number theory, computer algebra, computational statistics, computational physics and computational biology. Another way to classify algorithmics is to look at the generic tools and techniques that are independent of subject matter (to some degree). Along this line, we identify 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 algebraic computation, the complexity model takes each algebraic operation as a primitive while in the subfield of

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

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 has 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.

(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 $O = O(I)$.

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: 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 $(1, 2, 2, 5, 7)$.

Example: Decision or Recognition Problem: this refers to any input-output problem whose outputs have only two possible answers (0/1, Yes/No, Accept/Reject, etc). Such problems play an important role in complexity theory.

(B) Static 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.* This problem is a "static" preprocessing problem because the members of the set S does not change under the querying.

Example: 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: 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.

There are two algorithms in preprocessing problems: the algorithm to construct $D(S)$ and the algorithm to answer queries. They correspond to the two stages of preprocessing problems: first a **preprocessing part** to construct $D(S)$, then the **querying part** in which the data structure $D(S)$ is repeatedly 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 illustrates the larger class of **partial evaluation problems**. In such problems, we begin to 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.

(C) Dynamization and Online problems. If the input set S can be modified under queries, then we have a more general type of preprocessing problem which we call 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).*

Usually, updates are either 'insert an object' or 'delete an object' and D is a representation of the current set of objects. This is sometimes called the **set maintenance problem**. But in some applications, what is being maintained may not be obvious. 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: 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: 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.

Example: Online Scheduling Problems: In this case, each request r_i $(i = 1, \ldots, n)$ is a task that needs to be serviced. The requirement of the online problem is that the service decision s_i (to service it or not, when to serve it, etc) must be made before the next request is seen. We can view this as an optimization problem, where we are trying to maximize some function $f(s_1, s_2, \ldots, s_n)$ of the service decisions. There is a growing literature for a class of such algorithms that are "competitive" in the sense $f(s_1, \ldots, s_n)$ is at least some constant factor of the optimal solution $(s_1^*, \ldots, s_n^*)$.

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

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

Now that we know our quarry, we need to pick the appropriate tools. 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 data objects that it deals with,

(b) a collection of primitive operations on these objects, and

(c) rules for composing these 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. The Turing model "universal" in the sense that any other reasonable model can be embedded in this one. In algorithmics, non-universal models are sometimes preferred.

Example: For the sorting problem, two common computational models are the **comparison-tree model** and the **comparator circuit model**. In both models, the data objects are elements from a linear order. The comparison-tree model has only one primitive operation, viz., comparing the relative ordering of two elements $x : y$ resulting in one of two outcomes $x < y$ or $x \geq y$. 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.

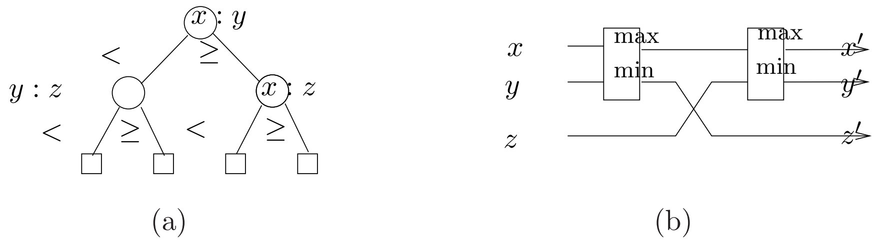

Figure 1: (a) A comparison-tree and (b) a comparator circuit

Example: 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.

Algorithms versus programs. To use a computational model to solve a given problem, we must make sure there is a match between the basic 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 such preliminaries, if a program A in a computational model solves a problem P, we call A an **algorithm for** P. Thus the term algorithm is a semantical concept, signifying a program in its relation to some problem. In contrast, programs are 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?

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 Turing machine model is an example of a **uniform model**. Another important computational model based on pointers is described at the end of Lecture V.4. There are ways to make non-uniform models "uniform" and therefore relate these two families of models.

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.

§4. The RAM model

In algorithmics, the most important computational model is the **random-access memory model** (RAM model). Actually, the RAM model is a generalization of a This will be the default model in this book unless otherwise noted. It is basically a simple, abstract assembly language. The RAM model is actually a generalization of an even simpler model called the **Register model**. We prefer to describe a register machine first.

(a) Data objects. We assume infinitely many **(storage) locations** which are numbered by an integer $0, \pm 1, \pm 2, \pm 3, \ldots$. Location 0 is special and is known as the **accumulator**. Each location can store an integer. Note that the integer is arbitrarily large.

(b) Primitive Operations. Each operation has the 4-field format

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

where $\langle LABEL\rangle$ and $\langle COMMENTS\rangle$ are arbitrary non-empty alphabetic strings – the square brackets indicate that these are optional fields. There is one argument $\langle ARG\rangle$ whose nature is determined by the $\langle OPERATOR\rangle$. The argument $\langle ARG\rangle$ is either an integer (denoted n below) or a label (denoted ℓ below). The contents of location n is denoted $c(n)$. The operators, their arguments and actions are specified in the following table:

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

(c) Semantics. A **register program** is any finite sequence of such primitive operations. The program begins executing the first instruction. Subsequently always execute the next instruction unless a successful jump operation is carried out in which case, the next instruction with this label is executed (multiply defined labels is no problem). The execution halts on encountering the HALT operation, or when there is no "next" instruction, or when the jump is to a non-existent label.

(d) Input/Output conventions. We assume that a finite number of locations is initialized with some encoding of the input, and the rest of the locations are initially zero. Some convention for the output can be assumed depending on the problem. As exercise, the reader may write a RAM program to compute the maximum of three numbers (convention: the input numbers are in the locations $1, 2, 3$ and the maximum value are to be output in location 0).

Random Access Feature. Notice that a register program can only access a fixed set of locations. These locations are often called "registers", hence the name for such machines. In order to allow a program to access an arbitrary number of locations, we need to allow indirect addressing. This is very simple: we now allow another form of integer arguments, denoted $@n$ where $n \in \mathbb{N}$. This means we are using the location $c(c(n))$ instead of $c(n)$. Hence we call $@n$ an "indirect argument". For instance, "GET @4" means $c(0) \leftarrow c(c(4))$. If location 4 contains 256, and location 256 contains -1, this means $c(0)$ is assigned the value -1. Similarly, "PUT @4" will place the value $c(0)$ into location 256. In conclusion: a **RAM program** is basically a register program in which we allow indirect addressing.

Variants. The instruction set is parsimonious (but not too severely so) by design. There are always trivial variants where we allow combinations of these primitive instructions. For instance, we can allow another kind of integer argument denoted $= n$. In this case the value n itself is being used. Hence we call this a "literal argument". Literal arguments are useful for PUT, but meaningless for GET instructions.

The above model may be called an **integer RAM model**. One generalization is where the locations can store an arbitrary real number, and possibly augmenting the primitive operations with other real functions (such as computing square roots). When we use a location n for indirect addressing ($@n$), we need to have some convention for handling the case where its contents $c(n)$ is not an integer. If there are no new primitive operations, then any non-integer values must come from the initial input.

Higher Level Languages. In practice, we may write our program using a more abstract language or a "higher level" language. Thus, we may use well-known constructs such as for-loops and if-then-else. Nevertheless, such extensions of the model is non-essential in the sense that any extended RAM program can be translated into a standard RAM program.

EXERCISES

Exercise 4.1: Write RAM algorithms for the following problems:
(a) Computing the GCD of two integers.
(b) Sorting a sequence of input numbers.
(c) Solve the ranking problem (§2). You need two algorithms, for preprocessing and for answering queries.
(d) Solve the dynamic ranking problem (§2). You need four algorithms here: to initialize an empty data structure, to insert, to delete, and to answer rank queries.
Be sure to state your input/output conventions. Also, describe any data structure you use. ◇

Exercise 4.2: Reduce the number of primitive operators (listed in table above) for our RAM model to a minimum. Show that your minimal RAM model can simulate our original RAM model. ♢

Exercise 4.3: Show how to implement higher level language constructs such as *for-loops, if-then-else, case-statements* in our RAM model. ♢

§5. Complexity model: how do we compare programs?

We now have a suitable computational model for solving our problem. How do we choose among different algorithms? 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

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

But using "sup" here is only one way to "aggregate" the set of numbers $\{T_A(I) : size(I) \leq n\}$. In general, we may apply a set-valued function G to the set,

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

For instance, G can be 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 consumption, 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.

There are complexity models especially in computational geometry in which an output size function is taken into account. This is the so-called **output-sensitive model**.

Example: 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 locations 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.

Static Complexity Measures. The **size** of a program in the RAM model may be taken to 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.

Exercises

Exercise 5.1: (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 location 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 5.2: Describe suitable complexity models for the the "space" resource for in integer RAM models. Give two versions, analogous to the unit time and logarithmic time versions. What about real RAM models? ◊

Exercise 5.3: With respect to the comparator circuit and tree program models in §3, describe suitable complexity models for each. ◊

§6. Algorithmic Tools: how to design algorithms

Now that we have some criteria to judge algorithms, we begin to design algorithms. There emerges several general paradigms of algorithms design. For instance:
(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)

There are many variations or refinements of these paradigms. E.g., A clever form of "divide-and-conquer" (called "marriage-before-dividing") leads to an output-sensitive convex hull algorithm [Kirkpatrick and Seidel]. 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: 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: 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[2] 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: 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: 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: 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).

§7. Analysis: how to evaluate algorithms

Having designed our algorithm A, we need to determine the complexity characteristics of A in our chosen complexity model. This constitutes the subject of **algorithmic analysis** which is a major part of this book. The mathematical tools for this depends on the algorithmic technique or data structure used. We give two examples.

Example: (Divide-and-conquer) If we use divide-and-conquer then it is likely we need to solve some recurrence equations. For instance, if a problem of size n is divided into 3 subproblems each of size $n/2$ and the cost of marrying the subproblems is $f(n)$ then we face the following recurrence equation:

$$T(n) = 3T(n/2) + f(n)$$

In our next lecture we study the solutions of such equations.

Example: (Amortization) If we employ certain data-structures that might be described as "lazy" then amortization analysis might be needed. Let us illustrate this with binary search trees. Here, it is known that the optimal solutions achieve logarithmic depth. Many known solutions can maintain trees of logarithmic depth, hence achieving logarithmic complexity *per operation*. But it may be advantageous to be lazy about maintaining this depth restriction: such laziness is rewarded by a simpler coding or programming effort. The

[2] A discipline of a different sort is called GIGO, or, garbage-in garbage-out. This is really a law of nature that no programming can change.

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 technique as well as an analysis methodology.

§8. Asymptotics: how robust is the model?

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

We started with a problem, selected a computational model and an associated complexity model, designed an algorithm and managed (being lucky) 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 invariance in our results.

This *robustness* or *invariance* issue is crucial if we are to expect some general validity to any of our analysis. There are many ways to achieve this: for instance we may state complexity functions up to constant powers: this is sometimes called "polynomial smearing" because all polynomial functions fall under the same class. This is *extremely* robust but alas, too drastic for most purposes. The most widely accepted procedure is to take two smaller steps: (1) we distinguish functions only up to multiplicative constants (e.g., $n/2$, n and $10n$ are indistinguishable), and (2) 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). These two decisions give us most of the robustness properties we desire, and are captured in the following language of asymptotics.

In the following, we simply say "function" to mean partial real functions,

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

We call f a **complexity function** provided f is defined on all sufficiently large natural numbers. We want to use complexity functions to quantify the complexity of our algorithms. Why do we consider partial functions? One reason is that 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. But the reason we prefer to use the entire set of reals for a domain is because we often use functions[3] such $n/2$ or $\sqrt{n}$, to bound our complexity functions. Also note that important real functions such as $\log n$ or $1/n$ are partial functions. *Unless otherwise noted, the designated variable in complexity functions, usually denoted 'n' or 'x', is assumed to range over real numbers.*

[3]We write $n/2$ to denote the function $f(n) = n/2$. In general, if we understand n to be the "designated variable", then we can write any expression $E(n)$ to denote the function $f(n) = E(n)$.

Domination. Given two functions, we say f **dominates** g, written

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

if $f(x) \geq g(x)$ holds for "x large enough" or, as we prefer to say, $f \geq g$ **eventually**. More precisely, this means there is some x_0 such that for all $x \geq x_0$, we have $f(x) \geq g(x)$ whenever both side are defined. To show the role of the x variable, we may also write

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

Clearly, domination is a transitive relation. Let us explain the "eventually" terminology in 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.).

For example, we say f is eventually non-negative if f dominates 0:

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

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

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

Thus means $f(x) = g(x)$ for sufficiently large x, whenever both sides are defined.

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 for which there exists a positive constant $C = C(g) > 0$ satisfying

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

Note that each function in $\mathcal{O}(f)$ dominates 0. In particular, $\mathcal{O}(f)$ is the empty set unless f dominates 0.

E.g., The set $\mathcal{O}(1)$ is the set of functions f that is dominated by some constant $C = C_f > 0$. Thus, $1 + \frac{1}{n}$ is a member of $\mathcal{O}(1)$ since it is dominated by any constant $C > 1$.

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

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

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

Relational form of the big-Oh notation. We write

$$f \preceq g \qquad \text{or} \qquad g \succeq f$$

if $f = \mathcal{O}(g)$. 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., $1 + \frac{1}{n} \prec n \prec n^2$.

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 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 partial real function then the following are $\mathcal{O}$-expressions:

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

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 $g \in \widetilde{F}$.

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

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

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:

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

Extended big-Oh notations. We introduce two 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,

$$O_A(n), \quad O_1(n^2) + O_2(n \log n).$$

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 occurences of an identically subscripted $\mathcal{O}$-notation are meant to refer to the same function.

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

$$n2^k = O_k(n), \quad n2^k = O_n(2^k)$$

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

$$g(n) = O_1(n \log n) = O_2(n^2)$$

where, it should be noted, the equalities here are true equalities of functions. Hence, with subscripted O-notations, we no longer feel it is an abuse to notations to write something like

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 $C > 0$,

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

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

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 $f = o(g)$.

Lower Bounds. Based on our asymptotic notations, we can specify lower bounds on a complexity function $f(n)$ in one of three form:

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

It is not hard to see that each form is less stringent than the previous. See Exercise for how these are used in practice.

Discussions. There is some debate over the best way to define the above concepts. Here we note just two alternatives:
1. Perhaps the most common definition follows Knuth [2, 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[4] between $1 + \mathcal{O}(1/n)$ and $1 - \mathcal{O}(1/n)$.
2. Again, we could have defined $\mathcal{O}(f)$ more simply to comprise those g such that for some $C > 0$, $g \leq C \cdot f$ (ev.). That is, we omit the requirement $g \geq 0$ (ev.) from our original definition. This definition is attractive because of its simplicity. But with this "simplified definition", $\mathcal{O}(f)$ contains arbitrarily negative functions. Thus, 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.

[4] 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 using the special notation $|\mathcal{O}|(f(n))$, etc.

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. If $|f - g| = \mathcal{O}(1)$, we cannot simply say "$f = g + \mathcal{O}(1)$" without further analysis, because the correct statement might be $f = g - \mathcal{O}(1)$.

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 "$f(n) \ge E$" where E is an $\mathcal{O}$-expression. The latter is not very meaningful. Hence we regard any use of $\le$ or $\ge$ symbols in $\mathcal{O}$-notations as illegitimate.

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 $g(n) \ne O(f(n))$ 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.

Exercises

Exercise 8.1: Assume $f(n) > 1$ (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 $f(n) > 1$ (ev.) is false. ♢

Exercise 8.2: Prove or disprove: $f = \mathcal{O}(1)^n$ iff $f = 2^{\mathcal{O}(n)}$. ♢

Exercise 8.3: 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 8.4: 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) $\lceil \log \log n \rceil! \le n$ (ev.). ♢

Exercise 8.5: 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 satisfy $f, g \ge 0$ (ev.).
(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) $f(n) = \Theta(f(n/2))$.

$\diamondsuit$

Exercise 8.6: Re-solve the previous exercise, assuming that f, g are unbounded and dominates 0. $\diamondsuit$

Exercise 8.7: 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) Again, what does it take to prove a lower bound of the form $T_A(n) = \Omega(f(n))$? $\diamondsuit$

Exercise 8.8: Show some examples where you might want to use "mixed" asymptotic expressions. $\diamondsuit$

Exercise 8.9: 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. $\diamondsuit$

§9. Two Dictums of Algorithmics

We state two principles in algorithmics. Appreciating them will help the student understand the motivation of many fundamental questions we ask.

(A) **Complexity functions are generally determined only up to Θ-order.** This simply formalizes 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" meaning $T(n) = \Theta(n)$. Then simple and local modifications to the algorithm should generally 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 can sometimes be silly. For instance, on inputs of moderate size, an algorithm may actually be inferior when compared to another asymptotically slower algorithm. For special problems, we might be interested in constant multiplicative factors. For instance, let $c > 0$ be the least constant such the median of n numbers can be found with $\leq cn$ comparisons. It is known that $c \geq 2$ [John W. John (1985)] and $c \leq 2.95$ [Zwick and Zwick (1994)].

(B) *Only problems with complexity that are polynomial-bounded are feasible. Moreover, there is considered to be an unbridgeable chasm 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. Of course, polynomial-bounded complexity may not exactly be practical. But exponential complexity is certainly out of the 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.

Despite the caveats, these two dictums turn out to be quite 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 a 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.

§10. Inherent complexity: lower bounds

Let P be a computational problem. Suppose that we have designed an algorithm for P and showed that it beats every known algorithm. But how do we know it would not be beaten in the future? This anxiety, common to all champions, may be abated somewhat if we understand something of the **inherent complexity** of P. It would certainly reassure us if we can show that our algorithm is "optimal" in this sense: for every algorithm B for problem P, $T_A = \mathcal{O}(T_B)$. If A is optimal and for every optimal algorithm B, we have $T_B = \Theta(T_A)$, then it is natural to define the "inherent complexity" T_P of P to be T_A. Clearly T_P is defined up to Θ-order. Trouble is, the inherent complexity T_P may not exist. They do exist in non-uniform complexity models, which is easily seen. For the sorting problem in the comparison model, it is known that $T_P(n) = n \log n$ and is achieved by several known algorithms. Such provably optimal non-linear time algorithms are extremely rare. Of course, linear time algorithms are usually optimal, but for trivial reasons.

Connection to complexity theory. Algorithmics is sometimes viewed as a subarea of complexity theory (algorithmics used to be called "concrete complexity" although this label has fallen out of use, perhaps because it sounds too much like a subfield of civil engineering). One view of complexity theory is that it is the study of inherent complexity. But in fact inherent complexity is just as important in algorithmics. The true distinction seems to be that algorithmics is the study of the inherent complexity of *individual problems* while complexity theory seeks to study properties of **classes of problems** (cf. [3, Preface]). Incidentally, the use of complexity classes leads to a way out of the dilemma of the missing optimal algorithms [3]. Because of its emphasis on individual problems, different machine models becomes interesting for algorithmics as they may be appropriate for different problems. But complexity theory prefers to use general-purpose machine models (Turing machines for example). Thus the comparison-tree model is extremely specialized but it is important for studying the complexity of problems on linear orders. A more general-purpose model would blur many distinctions in such problems which are of interest in algorithmics.

§A. APPENDIX: General Notations

We gather some general notations used in these lectures. It constitutes the minimal general background we need. More advanced concepts will be introduced at the places where we need them.

Definition. We write $X := ...Y...$ when defining a term X in terms of $\ldots Y \ldots$.

Numbers. Denote the set of natural numbers[5] 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 $\lceil -2.3 \rceil = -2$.

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.

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 exponentials $\exp_b(x)$ and logarithms $\log_b(x)$ to base $b > 0$ are more fully described in the appendix of lecture 2. In particular, $\lg x$ and $\ln x$ refers to logarithms to base 2 and base $e = 2.718...$, respectively. When the base b is not explicitly specified, it is assumed to be some $b > 1$.

Programs. We write programs in a pseudo-language with standard programming constructs such as if-then-else and while statements. Assignment to programming variables is written $x \leftarrow ...$, but we may also write $... \to x$ when convenient.

Logic, Proofs, Induction. The student should know basic propositional (or Boolean) logic. But mathematical facts goes beyond propositional logic. Here is an example of a mathematical statement, P.

P: For all real x and y, there exists a real z such that if $x < y$ then $x < z < y$.

The student is assumed to understand (parse) such statements. P happens to be true.

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

All mathematical statements are of this nature. In particular, such statements contain variables (such as x, y, z in P). Each variable has an implied or explicit range (x, y, z range over "real numbers"), and each variable is quantified ("for all" or "there exists"). 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**. Sometimes a variable is **free**, *i.e.*., not quantified. By convention, free variables are assumed to be universally quantified when we view the statement as a mathematical statement. E.g., Let $P(x, y)$ be the following statement.

$$P(x, y)\text{: there exists } z \text{ such that if } x < y \text{ then } x < z < y.$$

The variables x and y are free. As a mathematical statement, $P(x, y)$ is equivalent to P above.

Constructing proofs or providing counter examples to mathematical statements is a basic skill. Three kinds of proofs are absolutely fundamental: (i) case analysis, (ii) induction, and (iii) contradiction.

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[6] $|w|$. 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^*.

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 $E \subseteq V^2 = V \times V$. It is common to say "graph" when we mean bigraph or digraph; the context should make the intent clear. The edges of graphs are written '(u, v)' or 'uv' where u, v are vertices. Of course, in the case of bigraphs, $uv = vu$.

Often a graph $G = (V, E)$ comes with auxiliary data. For instance, a "weight" function $W : V \to \mathbb{R}$, a distinguished vertex $s \in V$, etc. We then attach such information to our specification of G and write

$$G = (V, E; W, s, \ldots).$$

Another common auxiliary data is a **vertex coloring** of G: this is just a function $C : V \to S$. Then $C(v)$ is called the **color** of $v \in V$. If $|S| = k$, we call C a k-coloring. An **edge coloring** is similarly defined.

We have terminology for some special classes of graphs: If E is the empty set, $G = (V, E)$ is called an **empty graph**. $K_n = (V, \binom{V}{2})$ 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 $n = |V_2|$.

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', then we call G' the V'**-induced subgraph** of G or G' is the **restriction** of G to V'. We may write $G|V'$ for G'.

[6] 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.

A **path** (from v_1 to v_k) is a sequence $(v_1, \ldots, v_k)$ of vertices such that (v_i, v_{i+1}) is an edge. A digraph path is a **cycle** if $v_1 = v_k$ and $k > 1$. In the case of bigraphs, the path is a **cycle** if $v_1 = v_k$, $k > 2$ and for all $1 < i < k$, $v_{i-1} \neq v_{i+1}$. 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 nodes 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 u, v are **adjacent** if (u, v) and (v, u) are edges. 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 node is the number of edges issuing (respectively) from and into that node. The **out-degree** (resp., **in-degree** of a digraph is the maximum of the out-degrees (resp., in-degrees) of its nodes. The nodes of out-degree 0 are called **sinks** and the nodes of in-degree 0 are called **sources**. The **degree** of a node in a bigraph is the number of adjacent nodes; the **degree** of a bigraph is the maximum of degrees of its nodes.

Trees. A connected acyclic bigraph is called a **free-tree**. A digraph such that there is a unique source node (called the **root**) and all the other nodes have in-degree 1, is called a **tree**. The sinks in a tree are called **leaves** or **external nodes** and non-leaves are called **internal nodes**. Note that 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.

INTRODUCE: ORDERED TREES and LABELED TREES. The latter is a generalization of the former. Binary trees as example of the latter.

Binary Trees. These trees are of major importance in computer science. A **binary tree** is a tree T of out-degree at most two and every edge is labeled either `Left` or `Right`, with the restriction that if a node has two outgoing edges, then these two edges have different labels. If there is a `Left` edge from v to u then we call u a **left child** of v; similarly for a **right child**. Note that if a node has only one child, then that child could be either a left or a right child.

We use u.`Left` and u.`Right` to denote the left and right child of u. In conventional programming languages, the u.`Left` and u.`Right` are often implemented as pointers. These pointers are pre-allocated to the node u. Hence, even if u does not have a left child, the pointer u.`Left` is present but is set to some special value called `nil`. In many applications, we further assume that each node has a pointer to its parent, and this is denoted u.`Parent`.

Any node v in T can be viewed as the root of another binary tree called the **subtree at** v. The subtree at the left child of v is called the **left subtree of** v; the left subtree of v is **empty** if v has no left child.

Similarly for **right subtree of** v.

There are 5 paths associated with a node u in a tree T:

1. The unique path from the root of T to u. NOTE: it is often convenient to assume that each node has a pointer to its parent (in practice, most binary trees do have such pointers). Following these parent pointers, we can then talk about the unique path **from** u **to the root**.
2. The path $\pi = (u_0, u_1, \ldots, u_\ell)$ where $u_0 = u$ and u_i is the left child of u_{i-1} $(i = 1, \ldots, \ell)$ and u_ℓ is a leaf. We call π the **left subpath** of u. We similarly define the **right subpath** of u.
3. The path $\pi = (u_0, u_1, \ldots, u_\ell)$ where $u_0 = u$, u_1 is the left child of u_0, and $(u_1, \ldots, u_\ell)$ is the right subpath of u_1. We call π the **left spine** of u. We similarly define the **right spine** of u.

In case u has no left child, its left subpath and left spines is the trivial path (u) of length 0. We often identify a binary tree T with its root u_0. So when we speak of the left subtree, right spine, etc., of T, we mean the left subtree, right spine, etc., of u_0.

A binary tree is **complete** if every leaf has the same depth. It is easy to see that if the depth is $d \geq 0$ then the tree has $2^{d+1} - 1$ nodes and 2^d leaves. A node in a binary tree is **full** if it has two children; a binary tree is said to be **full** if every non-leaf is full.

A family of binary trees is **balanced** if for each n, there are trees in the family with n nodes and every such tree has height $\mathcal{O}(\log n)$. The implicit constant in the $\mathcal{O}$-notation depends on the particular family.

We warn that there is an alternative view of binary trees which we call **extended binary trees**, to be introduced in lecture III.

Binary Search Trees. The most important use of binary trees are for storing and searching items. A **binary search tree** is a binary tree that in addition stores a **key** at each node. The key at node u is usually denoted $u.\mathtt{Key}$. Keys are elements from a totally ordered set so that we may compare them. The keys in a binary search tree need not be distinct but they must satisfy the **binary search tree property**: all the keys in the left (resp., right) subtree of a node v are less (resp., greater) than or equal to the key at v.

The nodes of a binary tree can be listed in one of three standard ways: **pre-order**, **in-order** or **post-order**. These correspond to listing a node (respectively) (i) *before* any of its descendents, (ii) *after* all nodes in its left subtree but *before* any of the nodes in its right subtree, and (iii) *after* all of its descendents. In-order is also called **symmetric order**. Node listing is also called **tree traversal**.

Exercises

Exercise A.1: The following is a useful result about iterated floors and ceilings. It shows that $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$, for instance.
(a) Let n, b be positive integers. Let $N_0 := n$ and for $i \geq 0$, $N_{i+1} := \lceil N_i/b \rceil$. Show that $N_i = \lceil n/b^i \rceil$. Similarly for floors. 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. ♢

Exercise A.2: Let x, a, b be positive real numbers. Show that $\lfloor x/ab \rfloor \ge \lfloor \lfloor x/a \rfloor /b \rfloor$. Is it true that they are equal? ♢

Exercise A.3: This is a basic result about binary trees. Show that every binary tree on $n \ge 1$ nodes has height at least $\lceil \lg(1+n) \rceil - 1$. Also show that this is tight for each n. ♢

Exercise A.4: (Erdös-Rado) Show that in any 2-coloring of the edges of the complete graph K_n, there is a monochromatic spanning tree of K_n. HINT: use induction. ♢

Exercise A.5: 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 $\left\lfloor \frac{n+1}{2} \right\rfloor$ 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 $\left\lfloor \frac{n+1}{2} \right\rfloor$ leaves. HINT: first show it when $n = 2^t - 1$. Alternatively, consider binary heaps. ♢

Exercise A.6:

(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. ♢

References

[1] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, 1990.

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

[3] Chee K. Yap. *Introduction to the Theory of Complexity Classes*. Oxford University Press (to appear), 1996. Available on request from author and from URL `ftp://cs.nyu.edu/pub/local/yap/complexity-bk`.