Slides for a Course on the Analysis and Design of Algorithms

Chapter 1: Fundamental Notions

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

1. Introduction

1.1 Problems and Algorithms

Abstractly, a *problem* is just a function

$$p : \text{Problem instances} \to \text{Solutions}$$

Example: Sorting a list of integers

Problem instances = lists of integers

Solutions = sorted lists of integers

$p : \ell \mapsto$ Sorted version of ℓ

An *algorithm* for p is a program which computes p.

There are four related issue which warrant consideration:

Issue 1: Characterize those problem instances which are solvable by algorithm.

- This is not a trivial question.
- **Church's Thesis:** There is an "upper bound" which is shared by the idealization of virtually all known programming languages in which unlimited memory is allowed.
- The detailed study of this issue is a subject for a course in the theory of computation.
- In this course, the focus will be upon problems which admit an algorithm. (*solvable problems*)

Issue 2: Characterize the amount of computing resources which a given algorithm A requires to solve a problem p.

Issue 3: Given a particular problem p which is solvable by algorithm, characterize how good the best possible algorithm for p can be, in terms of the amount of computing resources required.

Issue 4: Given a problem p, design good algorithms for p.

1.2 Key Conceptual Properties of Algorithms

There are four key conceptual properties of algorithms:

Finiteness: The algorithm must eventually halt for all specified input data.

Correctness: The algorithm must compute the correct function.

Definiteness: Each step of the algorithm must be precisely defined.

Effectiveness: The steps must be executable on a "physically realizable" computer.

- *Definiteness* and *effectiveness* are guaranteed when the algorithm is specified in a "real" programming language.
- *Finiteness* and *correctness* must be established in each case.
- With regard to correctness, there is some common terminology which is useful to know:

  Partial correctness: The algorithm performs the correct computation whenever it halts.

  Total correctness: The algorithm is partially correct and it always halts.

1.3 Algorithm Analysis

- The main focus of *algorithm analysis* in this course will be upon the "quality" of algorithms already known to be totally correct.

- This analysis proceeds in two dimensions:

  Time complexity: The amount of time which execution of the algorithm takes, usually specified as a function of the size of the input.

  Space complexity: The amount of space (memory) which execution of the algorithm takes, usually specified as a function of the size of the input.

- Such analyses may be performed both *experimentally* and *analytically*.

1.4 Designing Algorithms

In addition to analyzing given algorithms, it is important to study the process of designing good algorithms in the first place.

- There are two principal approaches to algorithm design.

  By problem: Study sorting algorithms, then scheduling algorithms, *etc.*

  By strategy: Study algorithms by *design strategy*. Examples of design strategies include:

  - Divide-and-conquer
  - The greedy method
  - Dynamic programming
  - Backtracking
  - Branch-and-bound

- In this course, algorithm design will be studied by strategy. This offers the following advantages:

  - It provides a common framework for analysis.
  - It provides insight into how algorithms for new problems may be designed.

1.5 Computational Models

- Church's Thesis establishes that *what* may be accomplished algorithmically is independent of the choice of computational model.
- The computational model nonetheless does affect *how* these things are accomplished, and so is central to the issue of algorithm design and performance analysis.
- The study in this course will be limited to *sequential* algorithms.
  - Most computers nowadays are *still* sequential.
  - A solid grounding in sequential algorithms is essential to the understanding of parallel algorithms.
  - The study of parallel algorithms is the subject of other courses.
- The computational model to be used is that of a "typical" imperative programming language (Algol, Pascal, C, Java, *etc.*)
- The details which make these languages different will not be relevant at the level of analysis and design which will be presented.

1.6 Some Basics of Algorithm Analysis

A formal measure of algorithm quality is needed.

Consider the simple example of Bubblesort; the following program sorts the n-element array $a[1..n]$ into nondecreasing order.

for $i \leftarrow 2$ **to** n **do**
  for $j \leftarrow n$ **downto** i **do**
    if $a[j-1] > a[j]$
      then $/* \; swap \; */$
        $\langle \; temp \leftarrow a[j-1]; \; a[j-1] \leftarrow a[j]; \; a[j] \leftarrow temp; \; \rangle$

Space requirements:

- This program requires $n+3$ integer storage locations.
  - n locations for the array.
  - One location for *temp*.
  - One location each for the loop variables i and j.

Caution:

- Not every (mathematical) integer fits in a computer word.
- Using standard binary representation, the (nonnegative) integer k requires $\max(1, \log_2(k))$ bits for storage, so in a uniform representation, to sort the integers $\{0, 1, .., k\}$, a storage size for integers of at least $\log_2(k)$ bits is required.
- Thus, a total of $(k+3) \cdot \log_2(k+3)$ bits of storage is required.
- The measurement of space for an algorithm depends upon what is admitted as an elementary data structure.
- In the study of *algorithm* complexity (but not the later study of *problem* complexity), the computer notion of a word-based integer will be regarded as an elementary data structure.

Time requirements:

First question: What is to be measured?

Experimental measurement: In *profiling*, the actual system time used is measured.

Formal analysis: In formal algorithm analysis, the number of *elementary instructions* which have been executed is measured.

- Time measurement in formal analysis thus depends upon what is admitted as an elementary operation.
- In the analyses of these notes, the basic operations of an imperative programming language are taken as elementary.
- Procedure calls are *not* elementary.
- These ideas will not be developed formally; that which is admitted as elementary will be obvious from the examples.

- Next, a closer look at the time complexity of bubblesort is made.

- If the list is totally sorted to begin with, then:
  - There are no variable assignments, other than those to the loop variables.
    - For the moment, assignments to loop variables and comparisons for loop termination will be ignored.
  - Excluding operations to manage loop indices, the number of comparisons is as follows:

$$\left.\begin{array}{ll} \text{1st pass:} & n-1 \\ \text{2nd pass:} & n-2 \\ \vdots & \vdots \\ (n-1)\text{st pass:} & 1 \end{array}\right\} = \sum_{i=1}^{n-1} i = \frac{n\cdot(n-1)}{2} = \frac{1}{2}\cdot(n^2-n).$$

- If the list is in reverse order to begin with, then:
  - The number of comparisons is the same as above.
  - Each comparison results in a swap.
    - Thus, there are $\frac{3}{2}\cdot(n^2-n)$ assignments.
- It is easy to see that:
  - the *best case* (fewest elementary operations) occurs when the list is totally ordered. ($\frac{1}{2}\cdot(n^2-n)$ total operations)
  - the *worst case* (most elementary operations) occurs when the list is in reverse order. ($2\cdot(n^2-n)$ total operations)

- It is also reasonable to ask about average complexity.

Question: Over what should the average be taken?

- In general, this question does not have an easy answer.
- Average complexity must be considered on a case-by-case basis, and will be considered in detail for select problems.
- For this algorithm, the number of comparison and assignment operations κ of any case is bounded by

$$\frac{1}{2} \cdot (n^2 - n) \leq \kappa \leq 2 \cdot (n^2 - n)$$

- Any measure of the average must therefore lie between these bounds.

Justification for ignoring loop assignments and comparisons:

- Loop initialization requires one assignment and one comparison..
- Each pass through a loop requires:
  - One assignment to increase or decrease the loop variable.
  - One comparison to decide whether or not to continue.
- The number of assignments and comparisons necessary to manage loop indices is as follows.

$$
\begin{aligned}
\overbrace{2+\sum_{i=2}^{n}(2}^{\text{outer loop}} \overbrace{+(2+\sum_{j=n}^{i}2))}^{\text{inner loop}} &= 2\cdot(\sum_{i=2}^{n}(1+(1+\sum_{j=n}^{i}1))) \\
&= 2\cdot(1+\sum_{i=2}^{n}(2+(n-i+1))) \\
&= 2\cdot(1+\sum_{i=2}^{n}(n-i+3)) \\
&= 2\cdot(1+\sum_{i=3}^{n+1}i) \\
&= 2\cdot(1-(1+2)+\sum_{i=1}^{n+1}i) \\
&= 2\cdot(-2+(n+1)\cdot n/2) \\
&= 2\cdot(-4+n^2+n) \\
&= 2\cdot(n^2+n-4)
\end{aligned}
$$

- The total complexity κ' is obtained by combining this result with the formula for κ on the previous slide.

$$\frac{1}{2} \cdot (n^2 - n) + n^2 + n - 4 \leq \kappa' \leq 2 \cdot (n^2 - n) + n^2 + n - 4$$

or

$$\frac{3}{2} \cdot n^2 + \frac{1}{2} \cdot n - 4 \leq \kappa' \leq 3 \cdot n^2 - n - 4$$

- With or without the loop overhead, the complexity is represented by a formula of the form

$$k_2 \cdot n^2 + k_1 \cdot n + k_0$$

with the k_i real constants and $k_2 > 0$.

- The general idea is to consider behavior:
  - only as $n \to \infty$, and
  - up to constant multiples.
- Since

$$\lim_{n \to \infty} \frac{k_2 \cdot n^2 + k_1 \cdot n + k_0}{n^2} = k_2$$

the complexity $k_2 \cdot n^2 + k_1 n + k_0$ is considered to be equivalent to the simpler formula n^2, and it is said that the *order* of the time complexity of the bubblesort algorithm is n^2.

- This idea will be formalized shortly.

Further remarks on complexity:

➢ In general, it is not the case that the best and worst case time complexities of an algorithm are of the same order.

Example: Improved bubblesort

$$
\begin{array}{l}
\mathit{swapflag} \leftarrow \mathit{true};\ i \leftarrow 2; \\
\textbf{while}\ \mathit{swapflag}\ \textbf{do} \\
\quad \langle\ \mathit{swapflag} \leftarrow \mathit{false}; \\
\quad\ \textbf{for}\ j \leftarrow n\ \textbf{downto}\ i\ \textbf{do} \\
\quad\quad \textbf{if}\ a[j-1] > a[j] \\
\quad\quad\ \textbf{then} \\
\quad\quad\quad \langle\ \mathit{temp} \leftarrow a[j-1];\ a[j-1] \leftarrow a[j]; \\
\quad\quad\quad\ a[j] \leftarrow \mathit{temp}; \\
\quad\quad\quad\ \mathit{swapflag} \leftarrow \mathit{true};\ \rangle \\
\quad\ i \leftarrow i+1; \\
\quad \rangle
\end{array}
$$

- In this version of bubblesort:
  - The worst case time complexity is of order n^2, as before.
  - The best case time complexity is of order n.

➢ The analyses presented here assume that integer comparison and assignment may be performed in constant time.

- This assumption is reasonable for integers of fixed storage size.
- It is not valid in situations in which integers of arbitrary size may be represented (*e.g.*, Common Lisp).
- In these notes, unless specifically stated to the contrary, it will be assumed that all integers are of fixed storage size.

1.7 Formalization of the Concept of Order

The next task is to formalize the notion of one function being more difficult to compute than another, subject to the following assumptions:

- The argument n defining the instance size is sufficiently large (asymptotic measure).
- Positive constant multipliers are ignored.

The following notation will be used throughout these notes.

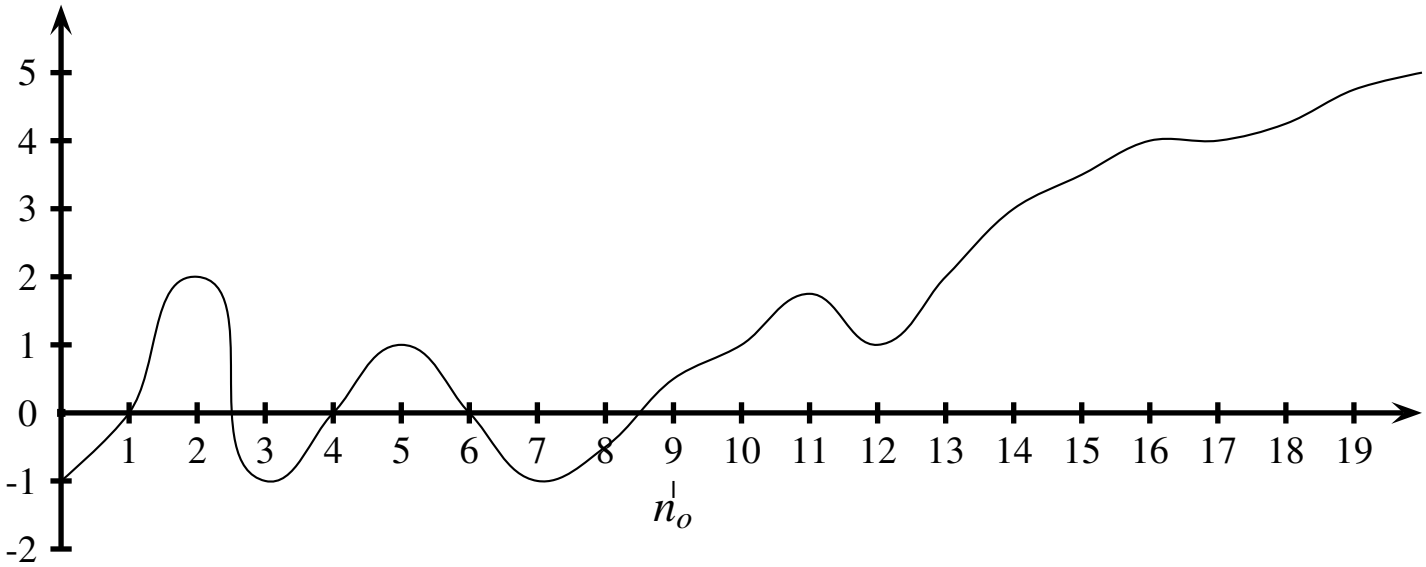
1.7.1 Notation for common sets of numbers

$$
\begin{array}{lcl}
\mathbb{N} & = & \text{natural numbers} = \{0, 1, 2, \ldots\} \\
\mathbb{N}^{>0} & = & \text{positive integers} = \mathbb{N} \setminus \{0\} = \{1, 2, \ldots\} \\
\mathbb{Z} & = & \text{integers} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
\mathbb{Q} & = & \text{rational numbers} \\
\mathbb{R} & = & \text{real numbers} \\
\mathbb{R}^{\geq 0} & = & \text{nonnegative real numbers} \\
\mathbb{R}^{>0} & = & \text{positive real numbers} \\
\mathbb{C} & = & \text{complex numbers}
\end{array}
$$

1.7.2 Definition Let $f : \mathbb{N} \to \mathbb{R}$. f is said to be *eventually nonnegative* (abbreviated e.n.) if there is an $n_o \in \mathbb{N}$ such that $f(n) \in \mathbb{R}^{\geq 0}$ for all $n \geq n_o$. The special notation

$$f : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R}$$

is used to denote that f is eventually nonnegative.



- "Physically meaningful" complexity functions will never be negative for usable arguments.
- However, $f(n)$ may not be meaningful for all values of n, and the negative values of $f(n)$ may correspond to values of n which are never used.
- In any case, behavior for values of n which are less than n_o is not significant for the asymptotic mathematical analysis.

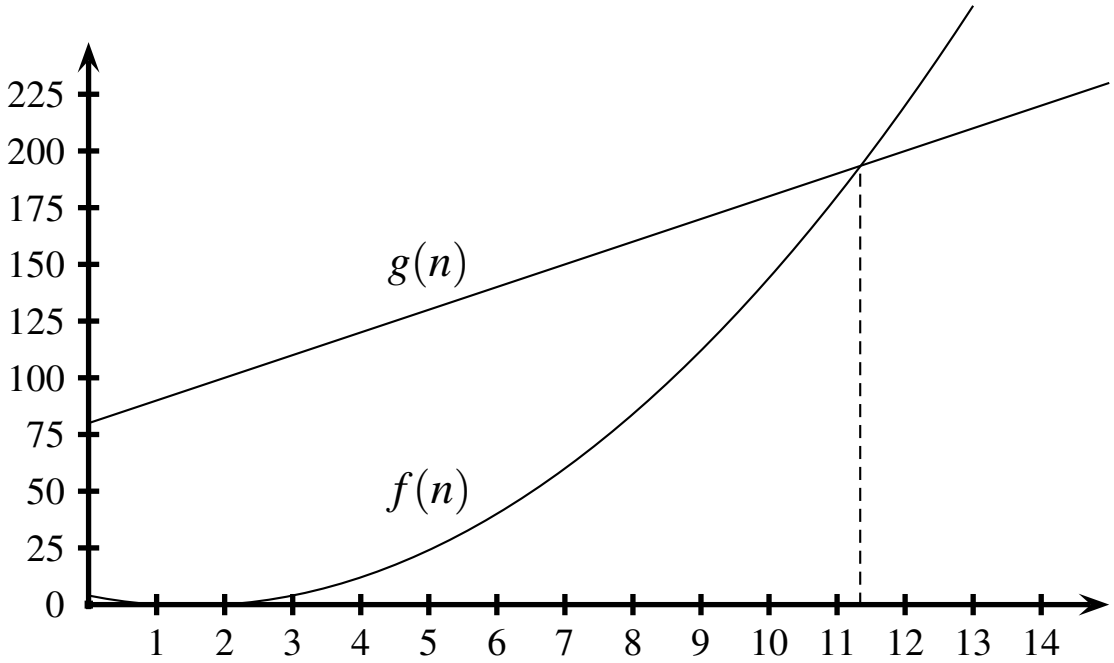
1.7.3 Definition Let $f : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R}$. Define

$$O(f) = \{g : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R} \mid (\exists n_o \in \mathbb{N})(\exists c \in \mathbb{R}^{>0})(\forall n \geq n_o)(g(n) \leq c \cdot f(n))\}$$

If $g \in O(f)$:

- It is said that "g is big-oh of f".
- It is customary to write $g = O(f)$, although $g \in O(f)$ is more consistent mathematically.
- The intuition is that g is "smaller" than f; *i.e.*, that g represents a lesser complexity.

1.7.4 Example Let $f(n) = 2 \cdot n^2 - 6 \cdot n + 4$;
$g(n) = 10 \cdot n + 80$.



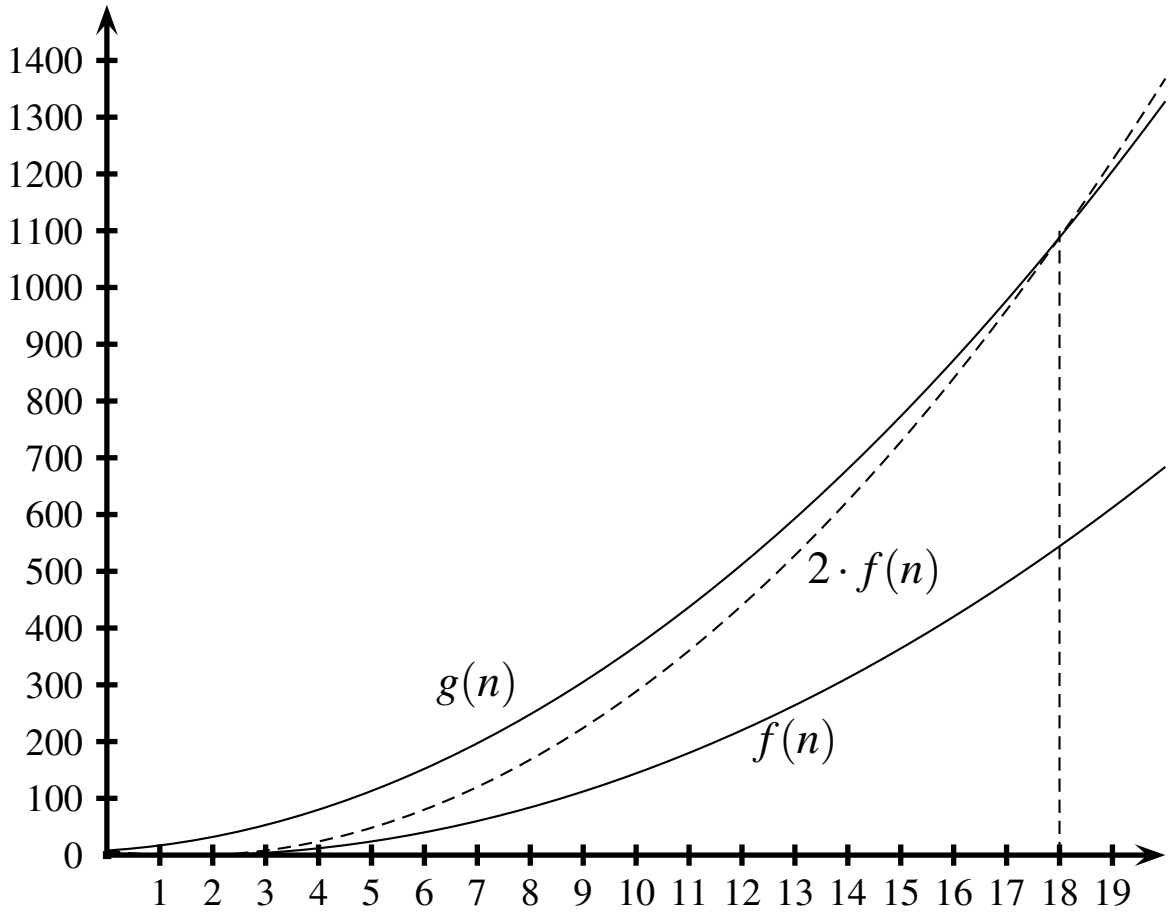
- To find the point of intersection, solve

$$2 \cdot n^2 - 6 \cdot n + 4 = 10 \cdot n + 80$$

- The positive solution is $n = 11.34$ (approximately).
- For $n \geq 12$, $g(n) \leq f(n)$.
- In the notation of 1.7.3, $n_o = 12$ and $c = 1$.
- Note that $f \notin O(g)$:
  - Suppose that $f(n) \leq c_1 \cdot g(n)$ for all $n \geq n_1$.
  - Then $2 \cdot n^2 + (-6 - 10 \cdot n) + (4 - 80 \cdot c_1) \leq 0$ for all $n \geq n_1$.
  - However, $2 \cdot n^2 + (-6 - 10 \cdot n) + (4 - 80 \cdot c_1)$ is clearly positive for large enough n.

- It is not always the case that the constant c may be chosen to be 1.

1.7.5 Example Let $f(n) = 2 \cdot n^2 - 6 \cdot n + 4$;
$g(n) = 3 \cdot n^2 + 6 \cdot n + 8$;



- $f(n) \leq g(n)$ for all $n \in \mathbb{N}$.
- $2 \cdot f(n) - g(n) = (4 \cdot n^2 - 12 \cdot n + 8) - (3 \cdot n^2 + 6 \cdot n + 8)$
  $= n^2 - 18 \cdot n$
  which is ≥ 0 for $n \geq 18$.
- Thus, $c = 2$, $n_o = 18$ works, in the notation of 1.7.3.
- In this case, $f \in O(g)$ also.

The following results strengthen the notion that whenever $g \in O(f)$, g is less complex than f.

1.7.6 Definitions

(a) Let ComplFn denote $\{f \mid f : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R}\}$. ComplFn is called the class of *basic complexity functions*.

(b) For $f, g \in$ ComplFn, define $g \leq_O f$ iff $g \in O(f)$.

(c) For $f, g \in$ ComplFn, define $g \equiv_O f$ iff $O(g) = O(f)$.

- Intuitively, $\leq_O$ is an order relation on ComplFn which captures the notion of one function defining a lesser complexity than another.
- However, $\leq_O$ is not a partial order, because loops of the form $f \leq_O g \leq_O f$ may exist.
- To obtain a partial order, it is necessary to group the functions in such loops into equivalence classes.

1.7.7 Proposition

(a) *The relation $\leq_O$ is a preorder; that is, it is reflexive and transitive. In other words:*

- *For all $f \in$ ComplFn, $f \in O(f)$.*
- *For all $f, g, h \in$ ComplFn, if $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.*

(b) *The relation $\equiv_O$ is an equivalence relation, and, furthermore, $f \equiv_O g$ iff $f \leq_O g$ and $g \leq_O f$.* □

Upon replacing $\leq$ with $\geq$ in $(g(n) \leq c \cdot f(n))$, the "dual" $\Omega_1(f)$ of $O(f)$ is obtained.

1.7.8 Definition Let $f : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R}$. Define

$$\Omega_1(f) = \{g : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R} \mid (\exists n_o \in \mathbb{N})(\exists c \in \mathbb{R}^{>0})(\forall n \geq n_o)(g(n) \geq c \cdot f(n))\}$$

If $g \in \Omega_1(f)$:

- It is said that "g is big-omega sub one of f".
- The intuition is that g is "larger" than f; *i.e.*, that g represents a greater complexity.

1.7.9 Observation *For any* $f, g \in \mathsf{ComplFn}$,

$$g \in O(f) \textit{ iff } f \in \Omega_1(g). \quad \square$$

1.7.10 Definition For $f \in \mathsf{ComplFn}$, define

$$f \in \Theta_1(g) \Leftrightarrow f \in O(g) \text{ and } f \in \Omega_1(g).$$

If $g \in \Theta_1(f)$:

- It is said that "g is big-theta sub one" of f".
- The intuition is that g, as a complexity measure, is "the same " as f.

1.7.11 Fact *$f \in \Theta_1(g)$ iff $f \equiv_O g$.* □

1.7.12 Proposition: polynomials and complexity *Let $f(n) = \sum_{i=0}^{m} a_i \cdot n^i$ and $g(n) = \sum_{i=0}^{\ell} b_i \cdot n^i$, with $a_m > 0$ and $b_\ell > 0$. In other words, let f and g be polynomials in the single variable n, with lead coefficient nonzero. Then*

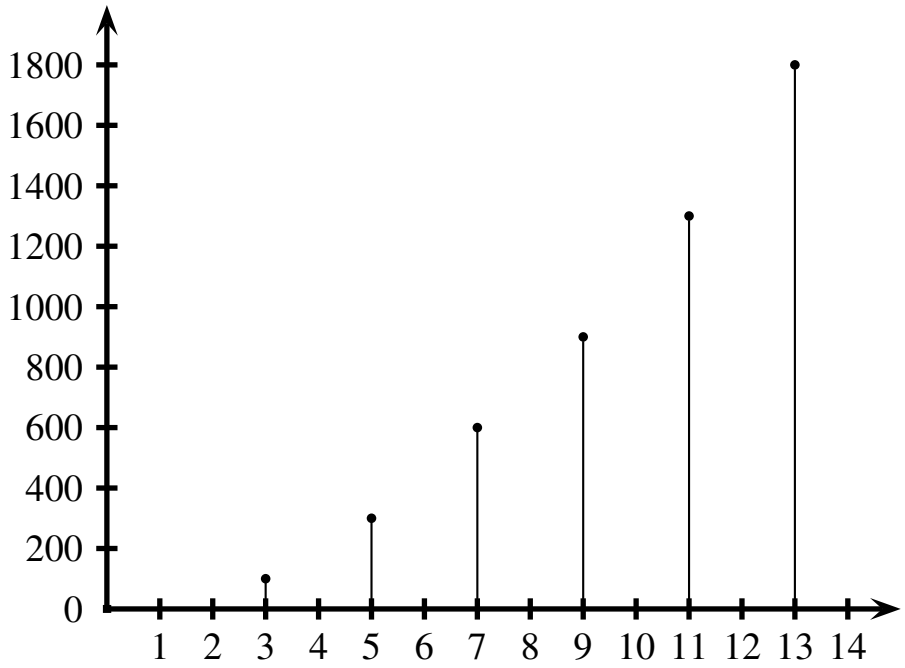
(a) $f \in \Theta_1(g) \Leftrightarrow m = \ell$.

(b) $f \in O(g) \Leftrightarrow m \leq \ell$.

(c) $f \in \Omega_1(g) \Leftrightarrow m \geq \ell$. □

There is an unfortunate drawback to the otherwise appealing nature of Θ_1 as a classification of complexity. The problem is best illustrated by example.

- Consider the problem of testing an integer for primality (*i.e.*, whether or not it can be factored into a product of two or more smaller integers).
- This problem is very difficult in general, and is used as the basis of most modern encryption algorithms.
- However, for even numbers, any algorithm can be made trivial, since no even number greater than two can be prime.
- Thus, any reasonable algorithm for testing primality will not be even $\Omega_1(\log(n))$.
- Clearly, this is not a reasonable characterization of the time requirements of the algorithm.



1.7.13 Definition Let $f : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R}$. Define

$$\Omega_2(f) = \{g : \mathbb{N} \xrightarrow{\text{e.n.}} \mathbb{R} \mid (\exists c > 0)(\forall m \in \mathbb{N})(\exists n \in \mathbb{N})((n > m) \wedge (g(n) \geq c \cdot f(n)))\}$$

If $g \in \Omega(f)$:

- It is said that "g is big-omega sub two of f".
- As with Ω_1, The intuition is that g is "larger" than f; *i.e.*, that g represents a greater complexity.
- The idea is that, up to constant multiple, g is bigger than f infinitely often.
- This characterization has the advantage that it recaptures the fact that algorithms for problems such as primality testing have high complexity.
- Unfortunately, it also has a strong disadvantage: the characterization is not symmetric, and does not result in equivalence classes of complexity functions.

The definition of Θ_2 is analogous to that of Θ_1.

1.7.14 Definition For $f \in \mathsf{ComplFn}$, define

$$f \in \Theta_2(g) \Leftrightarrow f \in O(g) \text{ and } f \in \Omega_2(g).$$

If $g \in \Theta_2(f)$:

- It is said that "g is big-theta sub two" of f".

The following example shows that Θ_2 is not symmetric.

1.7.15 Example Let $f(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}$
$g(n) = n^2$.
Then the following hold:
$f \in O(g)$ $f \in \Omega_2(g)$ $f \in \Theta_2(g)$ $f \notin \Omega_1(g)$ $f \notin \Theta_1(g)$
$g \notin O(f)$ $g \in \Omega_2(f)$ $g \notin \Theta_2(f)$ $g \in \Omega_1(f)$ $g \notin \Theta_1(f)$.

To close this subsection, the relationship of some common complexity classes is presented.

1.7.16 Proposition $O(1) \subsetneq O(\log(n)) \subsetneq O(n) \subsetneq (n \cdot \log(n)) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq \ldots \subsetneq O(n^k) \subsetneq O(2^n) \subsetneq O(3^n) \subsetneq \ldots \subsetneq O(k^n) \subsetneq O(n^n)$. □

- In the above $\subsetneq$ denotes that the inclusion is proper.

1.8 Recurrence Relations

Recurrence relations bear the same relationship to discrete problems (such as those which arise in algorithm analysis) as do differential equations to continuous-time problems.

In this section, some basic principles, as are applicable to algorithms analysis and design, are presented.

1.8.1 Motivating example Consider the following recursive program for computing the standard Fibonacci sequence.

function $fib(n : \mathbb{N})$
  $fib \leftarrow$**case** (n)
    $n = 0 : 0;$
    $n = 1 : 1;$
    $n > 1 : fib(n-1) + fib(n-2);$
  end case

Let $T(n)$ denote the time required to compute $fib(n)$. Then

$$\begin{aligned} T(0) &= k_1 \\ T(1) &= k_2 \\ T(n+2) &= T(n+1) + T(n) + k_3 \end{aligned}$$

with the k_i's constants. These are an instance of *linear recurrence relations*, the solution of which will now be examined in a general setting.

1.8.2 Definition A *homogeneous linear recurrence of order k* (with constant coefficients) is an equation of the form

$$a_k \cdot T(n+k) + a_{k-1} \cdot T(n+k-1) + \ldots + a_1 \cdot T(n+1) + a_0 \cdot T(n) = 0$$

with each $a_i \in \mathbb{R}$ and $a_k \neq 0$. More compactly, this recurrence may be written as

$$\sum_{i=0}^{k} a_i \cdot T(n+i) = 0 \qquad (*)$$

A *solution* to $*$ is a function $s : \mathbb{N} \to \mathbb{R}$ such that for all $n \in \mathbb{N}$,

$$\sum_{i=0}^{k} a_i \cdot s(n+i) = 0$$

With the equation $*$ is associated the following *characteristic polynomial* in the single variable x:

$$\sum_{i=0}^{k} a_i \cdot x^i$$

1.8.3 Fundamental Theorem of Algebra *Any polynomial of the form* $\sum_{i=0}^{k} a_i \cdot x^i$ *with each* $a_i \in \mathbb{R}$ *and* $a_k \neq 0$ *has a unique factorization of the form*

$$a_k \cdot (x - \xi_1) \cdot (x - \xi_2) \cdot \ldots \cdot (x - \xi_k)$$

with $\xi_i \in \mathbb{C}$ *for each i,* $1 \leq i \leq k$.

PROOF: Consult a good advanced text on algebra. □

The ξ_i are called the *roots* of the polynomial.

1.8.4 Theorem: solution of recurrences with distinct roots *Let*

$$\sum_{i=0}^{k} a_i \cdot T(n+i) = 0$$

be a homogeneous linear recurrence of order k, and suppose further that all k roots of its characteristic polynomial are distinct. Then any solution s of this recurrence has the general form

$$s(n) = \sum_{i=1}^{k} c_i \cdot \xi_i^n$$

with each $c_i \in \mathbb{C}$. *Here* ξ_i *is the* i^{th} *root of the characteristic polynomial* $\sum_{i=0}^{k} a_i \cdot x^i$.

PROOF: The proof of uniqueness is rather involved, and will not be presented here. However, it is straightforward and instructive to verify that s described above is indeed a solution. It suffices to note the following:

- $s(n) = \xi_j^n$ is a solution for any root ξ_i of the characteristic polynomial. (Just plug it in to the formula and reduce!)
- The linear sum of solutions of this form is also a solution because the equation is linear. □

1.8.5 Example Consider again the Fibonacci example of 1.8.1, and assume for the moment that $k_3 = 0$. The characteristic polynomial is then

$$x^2 - x - 1$$

whose roots are

$$\frac{1 \pm \sqrt{(-1)^2 + 4}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

Thus, the general form of the solutions to this recurrence is

$$s(n) = c_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$$

To determine the constants c_1 and c_2, the initial values $T(0) = k_1$ and $T(1) = k_2$ are used. Thus:

$$\begin{aligned} c_1 + c_2 &= k_1 \\ c_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right) + c_2 \cdot \left(\frac{1-\sqrt{5}}{2}\right) &= k_2 \end{aligned}$$

the solution of which is

$$\begin{aligned} c_1 &= \frac{\left(k_2 - k_1 \cdot \left(\frac{1-\sqrt{5}}{2}\right)\right)}{\sqrt{5}} \\ c_2 &= \frac{\left(-k_2 + k_1 \cdot \left(\frac{1+\sqrt{5}}{2}\right)\right)}{\sqrt{5}} \end{aligned}$$

If $k_3 \neq 0$, the following "shift and subtract" trick may be employed.

$$\begin{array}{rcrcrcl} T(n+3) & - & T(n+2) & - & T(n+1) & = & k_3 \\ & & T(n+2) & - & T(n+1) - T(n) & = & k_3 \\ \hline T(n+3) & - & 2 \cdot T(n+2) & & + T(n) & = & 0 \end{array}$$

The characteristic polynomial of the result is

$$x^3 - 2 \cdot x^2 + 1 = (x^2 - x - 1) \cdot (x - 1)$$

Thus, the solutions must be of the form

$$\begin{aligned} s(n) &= c_1 \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n + c_3 (1)^n \\ &= c_1 \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n + c_3 \end{aligned}$$

To solve for the three constants c_1, c_2, and c_3, one further initial condition is needed, such as

$$T(2) = k_1 + k_2 + k_3$$

The values of the c_i's can then be ground out as the solution of three linear equations in three variables.

In 1.8.4, the condition that all roots of the characteristic polynomial be distinct is mandated. Occasionally, cases in which two or more roots have the same value arise. The following theorem provides the means to handle such cases.

1.8.6 Theorem: solution of recurrences with repeated roots *Let*

$$\sum_{i=0}^{k} a_i \cdot T(n+i) = 0$$

be a homogeneous linear recurrence of order k, and let $\{\xi_1, \xi_2, \ldots, \xi_\ell\}$ *be the set of roots of the corresponding characteristic polynomial* $\sum_{i=1}^{k} a_i \cdot x^i$. *Suppose further that root* ξ_i *has multiplicity* α_i *(i.e.,* ξ_i *occurs* α_i *times). that all k roots of its characteristic polynomial are distinct. Then any solution s of this recurrence has the general form*

$$s(n) = \sum_{i=1}^{\ell} \sum_{m=0}^{\alpha_j - 1} c_i \cdot n^m \cdot \xi_i^n$$

with each $c_i \in \mathbb{C}$. *Here* ξ_i *is the* i^{th} *root of the characteristic polynomial* $\sum_{i=0}^{k} a_i \cdot x^i$.

PROOF: Omitted. □

1.8.7 Example Consider the following recurrence:

$$T(n+6)+8\cdot T(n+5)+19\cdot T(n+4)-11\cdot T(n+3)\\ -11\cdot T(n+2)-20\cdot T(n+1)-12\cdot T(n)=0$$

The characteristic polynomial is

$$x^6+8\cdot x^5+19\cdot x^4-11\cdot x^3-11\cdot x^2-20\cdot x-12\\ =(x-3)\cdot(x-2)^2\cdot(x-1)^3$$

Thus, solutions must be of the form

$$s(n)=c_1\cdot 3^n+c_2\cdot 2^n+c_3\cdot n\cdot 2^n+c_4+c_5\cdot n+c_6\cdot n^2$$

1.9 Inhomogeneous Recurrences

The technique described a the end of 1.8.5 shows how to deal with an inhomogeneous recurrence in which the additional term is a constant. However, more general cases occur in practice, and so it is useful to have more general techniques.

1.9.1 Theorem: solution of inhomogeneous recurrences *Let*

$$\sum_{i=0}^{k} a_i \cdot T(n+i) = b^n \cdot p(n)$$

be an inhomogeneous linear recurrence of order k in which b is a constant and $p(n)$ is a polynomial in n with real coefficients. The solutions of this recurrence are defined by the roots of the polynomial

$$\rho \cdot (x-b)^{d+1}$$

in which ρ is the characteristic polynomial of the corresponding homogeneous recurrence and d is the degree of the polynomial $p(n)$.

PROOF: Omitted. □

1.9.2 Example For the inhomogeneous component of 1.8.5,

$$\begin{aligned} \rho &= x^2 - x - 1 \\ b^n \cdot p(n) &= k_3 \end{aligned}$$

whence $b = 1$, $p(n) = k_3$, and $d = 0$. Thus, the characteristic polynomial is

$$(x^2 - x - 1) \cdot (x-1)$$

as before.

1.10 Geometric Recurrences

1.10.1 Example Consider a recurrence of the following form.

$$T(n) - 2 \cdot T(n/2) = k \cdot n$$

in which k is a constant. As it stands, this recurrence is not linear. However, it may be transformed into a linear one upon performing the substitution $n \rightsquigarrow 2^m$. The resulting recurrence is

$$T(2^m) - 2 \cdot T(2^{m-1}) = k \cdot 2^m$$

Next, define $\hat{T}(m) = T(2^m)$. The recurrence may then be written as

$$\hat{T}(m) - 2 \cdot \hat{T}(m-1) = k \cdot 2^m$$

which has the characteristic equation

$$(x-2) \cdot (x-2)$$

and hence solutions of the form

$$c_1 \cdot 2^m + c_2 \cdot m \cdot 2^m$$

However, $n = 2^m$; *i.e.*, $m = \log_2(n)$, so *for a power of two*, the solutions have the form

$$s(n) = c_1 \cdot n + c_2 \cdot n \cdot \log_2(n)$$