Ákos Seress
The Ohio State University

Permutation Group Algorithms

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typeface Times 10/13 pt. *System* $\LaTeX\,2_\varepsilon$ [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data available

ISBN 0 521 66103 X hardback

Contents

1
Introduction

Computational group theory (CGT) is a subfield of symbolic algebra; it deals with the design, analysis, and implementation of algorithms for manipulating groups. It is an interdisciplinary area between mathematics and computer science. The major areas of CGT are the algorithms for finitely presented groups, polycyclic and finite solvable groups, permutation groups, matrix groups, and representation theory.

The topic of this book is the third of these areas. Permutation groups are the oldest type of representations of groups; in fact, the work of Galois on permutation groups, which is generally considered as the start of group theory as a separate branch of mathematics, preceded the abstract definition of groups by about a half a century. Algorithmic questions permeated permutation group theory from its inception. Galois group computations, and the related problem of determining all transitive permutation groups of a given degree, are still active areas of research (see [Hulpke, 1996]). Mathieu's constructions of his simple groups also involved serious computations.

Nowadays, permutation group algorithms are among the best developed parts of CGT, and we can handle groups of degree in the hundreds of thousands. The basic ideas for handling permutation groups appeared in [Sims, 1970, 1971a]; even today, Sims's methods are at the heart of most of the algorithms.

At first glance, the efficiency of permutation group algorithms may be surprising. The input consists of a list of generators. On one hand, this representation is very efficient, since a few permutations in S_n can describe an object of size up to $n!$. On the other hand, the succinctness of such a representation $G = \langle S \rangle$ necessitates nontrivial algorithms to answer even such basic questions as finding the order of G or testing membership of a given permutation in G.

Initially, it is not even clear how to prove in polynomial time in the input length that a certain permutation g is in G, because writing g as a product of the given generators S for G may require an exponentially long word. Sims's seminal idea was to introduce the notions of base and strong generating set. This data structure enables us to decide membership in G constructively, by writing any given element of G as a short product of the strong generators. The technique for constructing a strong generating set can also be applied to other tasks such as computing normal closures of subgroups and handling homomorphisms of groups. Therefore, a significant part of this book is devoted to the description of variants and applications of Sims's method.

A second generation of algorithms uses divide-and-conquer techniques by utilizing the orbit structure and imprimitivity block structure of the input group, thereby reducing the problems to primitive groups. Although every abstract group has a faithful transitive permutation representation, the structure of primitive groups is quite restricted. This extra information, partly obtained as a consequence of the classification of finite simple groups, can be exploited in the design of algorithms.

We shall also describe some of the latest algorithms, which use an even finer divide-and-conquer technique. A tower of normal subgroups is constructed such that the factor groups between two consecutive normal subgroups are the products of isomorphic simple groups. Abelian factors are handled by linear algebra, whereas the simple groups occurring in nonabelian factors are identified with standard copies of these groups, and the problems are solved in the standard copies. This identification process works in the more general black-box group setting, when we do not use the fact that the input group is represented by permutations: The algorithms only exploit the facts that we can multiply and invert group elements and decide whether two group elements are equal. This generality enables us to use the same algorithms for matrix group inputs. Computations with matrix groups is currently the most active area of CGT.

Dealing with permutation groups is the area of CGT where the complexity analysis of algorithms is the most developed. The initial reason for interest in complexity analysis was the connection of permutation group algorithms with the celebrated graph isomorphism problem. The decisive result in establishing the connection is the polynomial-time algorithm in [Luks, 1982] for testing isomorphism of graphs with bounded valence, where the isomorphism problem is reduced to finding setwise stabilizers of subsets in the permutation domain of groups with composition factors of bounded size. This paper not only established a link between complexity theory and CGT but provided new methodology for permutation group algorithms.

Up until the end of the 1980s, permutation group algorithms were developed in two different contexts. In one of these, the primary goal was efficient implementation, to handle the groups occurring in applications. In the other context, the main goal was the rigorous asymptotic analysis of algorithms.

Algorithms for numerous tasks were developed separately in the two contexts, and the two previous books on permutation group algorithms reflect this division: [Butler, 1991] deals mostly with the practical approach, whereas [Hoffmann, 1982] concentrates on the asymptotic analysis. In the past decade, a remarkable convergence of the approaches occurred, and algorithms with fast asymptotic running times that are suitable for implementation were developed. The main purpose of this book is to describe this new development. We consider the interaction of theory and implementation to be of great importance to each side: Symbolic algebra can benefit considerably by the influx of ideas of algorithmic complexity theory and rigorous asymptotic analysis; conversely, the implementations help demonstrate the power of the asymptotic paradigm, which is at the foundation of the theory of computing.

The major theme of this book is the description of nearly linear-time algorithms. These are the algorithms representing the convergence of theoretical and practical considerations. Their running time is $O(n|S| \log^c |G|)$ for input groups $G = \langle S \rangle \leq S_n$; in particular, in the important subcase of small-base groups, when $\log |G|$ is bounded from above by a polylogarithmic function of n, the running time is a nearly linear, $O(N \log^c N)$, function of the input length $N = n|S|$. The category of small-base groups includes all permutation representations of finite simple groups except the alternating ones and all primitive groups that do not have alternating composition factors in their socle. Most practical computations are performed with small-base input groups.

Quite different methods give the asymptotically fastest solutions for computational problems in large-base groups, where $\log |G|$ is bounded only by $\log n!$. Most of these algorithms have not yet been implemented. We shall also describe backtrack methods, which are the practical algorithms for problems with no known polynomial-time solutions. For small-base input groups, backtrack methods may be practical in groups of degree in the tens of thousands.

Our main goal is to present the mathematics behind permutation group algorithms, and implementation details will be mostly omitted. We shall give details only in the cases where the implemented version differs significantly from the one described by the theoretical result or when the reason for the fast asymptotic running time is a nontrivial data structure. Most of the algorithms described in this book have been implemented in the GAP system [GAP, 2000], which, along with its source code, is freely available. GAP code is written in a high-level, Pascal-like language, and it can be read as easily as the customary pseudocode in other books and articles on computational group theory. The addresses of ftp servers for GAP can be obtained from the World Wide Web page

`http://www-gap.dcs.st-and.ac.uk/~gap.`

The other large computer algebra system particularly suitable for computations with groups is MAGMA (see [Bosma et al., 1997]). The World Wide Web page

```
http://www.maths.usyd.edu.au:8000/u/magma
```

describes how to access MAGMA on a subscription basis.

Acknowledgments

The writing of this book began in 1993, on the suggestion of Joachim Neubüser, who envisioned a series of books covering the major areas of computational group theory. The fact that the writing is finished in less than a decade is in no small part the consequence of my wife Sherry's continuous encouragement. I am thankful to both of them and to the editors of Cambridge University Press for their patience. During this period, I was partially supported by the National Science Foundation.

Alexander Hulpke, William Kantor, Joachim Neubüser, Cheryl Praeger, Charles Sims, and Leonard Soicher read parts of the manuscript, and their comments improved the presentation significantly. I am especially indebted to William Kantor and Joachim Neubüser for their help.

1.1. A List of Algorithms

In this book, most algorithms are described in the proofs of theorems or just in the narrative, without any display or pseudocode. Whenever it is possible, algorithms given in the narrative are preceded by a centered paragraph header. The following list serves as a reference guide; it is organized roughly along the lines of the lists in Sections 3.1 and 3.3. The input is a permutation group $G \leq \mathrm{Sym}(\Omega)$.

- Orbit of some $\alpha \in \Omega$: Section 2.1.1; in particular, Theorem 2.1.1
- Blocks of imprimitivity
  - (i) A minimal nontrivial block: Section 5.5.1 (algorithm MinimalBlock)
  - (ii) The minimal block containing a given subset of Ω: Section 5.5.2
- Shallow Schreier tree construction
  - (i) Deterministic: Lemma 4.4.2, Remark 4.4.3, Lemma 4.4.8
  - (ii) Las Vegas: Theorem 4.4.6, Remark 4.4.7
- Strong generating set construction
  - (i) Deterministic: Section 4.2 (Schreier–Sims algorithm), Theorem 5.2.3 (with known base), Section 7.1 (for solvable groups), Theorem 10.1.3 (stored in a labeled branching)
  - (ii) Monte Carlo: Section 4.5, Theorems 5.2.5 and 5.2.6, Lemma 5.4.1, Section 10.3

(iii) Heuristic: Section 4.3 (random Schreier–Sims algorithm)
(iv) GAP implementation: Section 4.5.1, Remark 5.2.7

- Strong generating set verification
  (i) Deterministic: Section 8.1 (Schreier–Todd–Coxeter–Sims algorithm), Section 8.2 (Verify routine)
  (ii) Monte Carlo: Lemma 4.5.6
  (iii) Las Vegas: Theorem 8.3.1
- Membership test (sifting): Section 4.1
- Reduction of the size of generating sets
  (i) Strong generators, deterministic: Lemma 4.4.8, Exercise 4.7
  (ii) Arbitrary generators, Monte Carlo: Lemma 2.3.4, Theorem 2.3.6
- Random element generation
  (i) With an SGS: Section 2.2, first paragraph
  (ii) Without an SGS: Section 2.2 (random walk on a Cayley graph, product replacement algorithm)
  (iii) In alternating and symmetric groups: Exercises 2.1 and 2.2
- Isomorphism with other representations
  (i) With a black-box group: Section 5.3
  (ii) Solvable groups, with a power-commutator presentation: Section 7.2
  (iii) A_n and S_n, with natural action: Theorem 10.2.4
  (iv) $\mathrm{PSL}_d(q)$, with the action on projective points: Section 5.3
- Operations with base images and with words in generators: Lemmas 5.2.1, 5.2.2, and 5.3.1
- Base change (transposing and conjugating base points, deterministic and Las Vegas algorithms): Section 5.4, Exercise 5.5
- Presentations: Section 7.2 (for solvable groups), Section 8.1, Exercise 5.2, Theorem 8.4.1
- Pointwise stabilizer of a subset of Ω: Section 5.1.1
- Handling of homomorphisms (kernel, image, preimage)
  (i) Transitive constituent and block homomorphisms: Section 5.1.3
  (ii) General case: Section 5.1.2
- Closure for G-action, normal closure
  (i) With membership test in substructures, deterministic: Sections 2.1.2 and 5.1.4, Lemma 6.1.1
  (ii) Without membership test, Monte Carlo: Theorems 2.3.9 and 2.4.5
- Commutator subgroup computation, derived series, lower central series
  (i) With membership test in substructures, deterministic: Sections 2.1.2 and 5.1.4
  (ii) Without membership test, Monte Carlo: Theorems 2.3.12 and 2.4.8
- Upper central series in nilpotent groups: Section 7.4.2
- Solvability test: Sections 2.1.2, 5.1.4, and 7.1

- Nilpotency test: Sections 2.1.2, 5.1.4, and 7.4.1
- Subnormality test: Section 2.1.2
- Commutativity test (Monte Carlo): Lemma 2.3.14
- Regularity test: Exercises 5.12–5.15
- Center: Section 6.1.3
- Permutation representation with a normal subgroup N in the kernel: Lemmas 6.2.2 and 6.2.4, Theorem 6.3.1 (if N is abelian)
- Composition series
  - (i) Reduction to the primitive group case: Section 6.2.1
  - (ii) Finding normal subgroups in various types of primitive groups (Monte Carlo): Sections 6.2.3 and 6.2.4
  - (iii) Verification of composition series, if SGS is known: Section 6.2.6
  - (iv) GAP implementation: Section 6.2.5
  - (v) Composition series without the classification of finite simple groups: Section 6.2.6
- Chief series: Section 6.2.7
- Sylow subgroups and Hall subgroups in solvable groups: Section 7.3.1, Exercise 7.5 (Theorem 7.3.3 for conjugating Sylow subgroups)
- Core of a subnormal subgroup: Section 6.1.5
- p-core and solvable radical: Section 6.3.1
- Backtrack, general description: Section 9.1 (traditional), Section 9.2 (partition backtrack)
- Setwise stabilizer of a subset of Ω: Section 9.1.2, Example 2
- Centralizer
  - (i) In the full symmetric group: Section 6.1.2
  - (ii) Of a normal subgroup: Section 6.1.4
  - (iii) General case: Section 9.1.2, Example 1
- Intersection of groups: Corollary 6.1.3 (if one of the groups normalizes the other), Section 9.1.2, Example 3 (general case)
- Conjugating element: Section 9.1.2, Example 4
- Conjugacy classes: Section 7.3.2 (in solvable groups), Section 9.4 (general case)
- Normalizer: Section 9.3

1.2. Notation and Terminology

We assume that the reader is familiar with basic notions concerning groups covered in introductory graduate courses and with elementary probability theory. A background area with which we do *not* suppose reader familiarity is the detailed properties of finite simple groups, and the occasional references to these properties can be ignored without impeding understanding of the subsequent material. However, readers interested in further research in permutation group algorithms are strongly advised to acquire knowledge of groups of Lie type. One of the current largest obstacles, both in the permutation group and matrix group setting, is our inability to exploit algorithmically properties of exceptional groups of Lie type.

The required (minimal) background material about permutation groups can be found for example in the first chapter of the recent books [Dixon and Mortimer, 1996] and [Cameron, 1999]. Here we only summarize our notation and terminology. In this book, *all groups are finite*.

All statements (i.e., theorems, lemmas, propositions, corollaries, and remarks) are numbered in a common system. For example, Theorem X.Y.Z denotes the Zth statement in Chapter X, Section Y, if this statement happens to be a theorem. Definitions are just part of the text and are not displayed with a number. Any unknown items (hopefully) can be found in the index. In the index, boldface type is used for the page number where an item or notation is defined. There are exercises at the end of some chapters, numbered as Exercise X.Y in Chapter X. A third numbering system is used for the displayed formulas, in the form (X.Y) in Chapter X.

1.2.1. Groups

If G is a group and $S \subseteq G$ then we denote by $\langle S \rangle$ the subgroup generated by S. We write $H \leq G$ to indicate that H is a subgroup of G and $H \lneqq G$ if $H \leq G$ and $H \neq G$. If H is isomorphic to a subgroup of G then we write $H \lesssim G$. The symbol $|G : H|$ denotes the number $|G|/|H|$, and $H \lhd G$ denotes that H is normal in G. A subgroup $H \leq G$ is *subnormal* in G, in notation $H \lhd\lhd G$, if there exists a chain of subgroups $H = H_0 \lhd H_1 \lhd \cdots \lhd H_k = G$. If $N \lhd G$ and $H \leq G$ such that $N \cap H = 1$ and $G = NH$ then we call H a *complement* of N in G.

The group of automorphisms, outer automorphisms, and inner automorphisms of G are denoted by $\mathrm{Aut}(G)$, $\mathrm{Out}(G)$, and $\mathrm{Inn}(G)$, respectively. We say that *G acts on a group H* if a homomorphism $\varphi : G \to \mathrm{Aut}(H)$ is given. If φ is clear from the context, for $g \in G$ and $h \in H$ we sometimes denote $\varphi(g)(h)$, the image of h under the automorphism $\varphi(g)$, by h^g. If G acts on H and $U \subseteq H$ then $U^G := \{U^g \mid g \in G\}$ is the *orbit* of U under the G-action, and $\langle U^G \rangle$ is the *G-closure* of U. In the special case $H = G$, the group $\langle U^G \rangle$ is called the *normal closure* of U. For $U \leq H$, $C_G(U) := \{g \in G \mid (\forall u \in U)(u^g = u)\}$ is the *centralizer* of U in G and $N_G(U) := \{g \in G \mid U^g = U\}$ is the *normalizer* of U in G. In particular, $Z(G) := C_G(G)$ is the *center* of G.

The *commutator* of $a, b \in G$ is $[a, b] := a^{-1}b^{-1}ab$ and the *conjugate* of a by b is $a^b := b^{-1}ab$. For $H, K \leq G$, the *commutator* of H and K is defined as $[H, K] := \langle [h, k] \mid h \in H, k \in K \rangle$. In particular, $[G, G]$, also called the *derived subgroup* of G, is denoted by G'. A group G is *perfect* if $G = G'$. The *derived series* of G is the sequence $D_0 \geq D_1 \geq \cdots$ of subgroups of G, defined recursively by the rules $D_0 := G$ and $D_{i+1} := D_i'$ for $i \geq 0$. The *lower central series* $L_0 \geq L_1 \geq \cdots$ of G is defined as $L_0 := G$ and $L_{i+1} := [L_i, G]$ for $i \geq 0$. The *upper central series* $Z_0 \leq Z_1 \leq \cdots$ of G is defined as $Z_0 := 1$ and Z_{i+1} is the preimage of $Z(G/Z_i)$ in G for $i \geq 0$. A group G is called *solvable* if $D_m = 1$ for some m, and it is called *nilpotent* if $L_m = 1$ or $Z_m = G$ for some m.

The *direct product* of groups $A_1, \ldots, A_m$ is denoted by $A_1 \times \cdots \times A_m$ or by $\prod_{i=1}^m A_i$. For $i = 1, 2, \ldots, m$, the *projection function* $\pi_i \colon A_1 \times \cdots \times A_m \to A_i$ is defined by the rule $\pi_i \colon (a_1, \ldots, a_m) \mapsto a_i$. A group $H \leq A_1 \times \cdots \times A_m$ is a *subdirect product* of the A_i if all functions π_i restricted to H are surjective, i.e., $\{\pi_i(h) \mid h \in H\} = A_i$.

For $H \leq G$, a *transversal* G mod H is a set of representatives from the right cosets of H in G. For a fixed transversal T and $g \in G$, we denote by $\bar{g}$ the coset representative in T such that $g \in H\bar{g}$. Unless stated explicitly otherwise, cosets always mean *right* cosets.

If Σ is any collection of simple groups, $O_\Sigma(G)$ denotes the largest normal subgroup of G such that each composition factor of $O_\Sigma(G)$ is isomorphic to a member of Σ, and $O^\Sigma(G)$ denotes the smallest normal subgroup of G such that each composition factor of $G/O^\Sigma(G)$ is isomorphic to a member of Σ. In particular, if Σ consists of a single group of prime order p then $O_\Sigma(G)$ is denoted by $O_p(G)$; this is the largest normal p-subgroup, the *p-core*, of G. When Σ consists of all cyclic simple groups, $O_\Sigma(G)$ is denoted by $O_\infty(G)$; this is the largest solvable normal subgroup, the *solvable radical* of G. Similarly, $O^\infty(G)$ denotes the smallest normal subgroup of G with solvable factor group and it is called the *solvable residual* of G. For $H \leq G$, $\mathrm{Core}_G(H) := \bigcap\{H^g \mid g \in G\}$ is the largest normal subgroup of G contained in H; it is the kernel of the permutation representation of G on the (right) cosets of H. The *socle* of G is the subgroup of G generated by all minimal normal subgroups of G and is denoted by $\mathrm{Soc}(G)$.

The cyclic group of order n is denoted by C_n. The group of invertible $d \times d$ matrices over the q-element field $\mathrm{GF}(q)$ is denoted by $\mathrm{GL}_d(q)$. Similar notation is used for the other classical matrix groups of Lie type and for their projective factor groups: $\mathrm{SL}_d(q)$, $\mathrm{PSL}_d(q)$, and so on. The unitary groups $\mathrm{GU}_d(q)$, $\mathrm{SU}_d(q)$, and $\mathrm{PSU}_d(q)$ are defined over $\mathrm{GF}(q^2)$. For exceptional groups of Lie type, we use the Lie-theoretic notation ${}^2B_2(q)$, ${}^2G_2(q)$, and so on. As mentioned earlier, no detailed knowledge of the groups of Lie type is required in this book.

1.2.2. Permutation Groups

We shall use the cycle notation for permutations, and the identity permutation is denoted by $()$. The group of all permutations of an n-element set Ω is denoted $\mathrm{Sym}(\Omega)$, or S_n if the specific set is inessential. Subgroups of S_n are the *permutation groups* of *degree n*. We use lowercase Greek letters to denote elements of Ω; lower- and uppercase italics denote elements and subgroups of S_n, respectively. For $\alpha \in \Omega$ and $g \in \mathrm{Sym}(\Omega)$, we write α^g for the image of α under the permutation g. The alternating group on Ω is denoted by $\mathrm{Alt}(\Omega)$ (or A_n). The *support* of $g \in \mathrm{Sym}(\Omega)$, denoted by $\mathrm{supp}(g)$, consists of those elements of Ω that are actually displaced by g: $\mathrm{supp}(g) = \{\omega \in \Omega \mid \omega^g \neq \omega\}$. The set of fixed points of g is defined as $\mathrm{fix}(g) := \Omega \backslash \mathrm{supp}(g)$. The *degree* of g is $\deg(g) = |\mathrm{supp}(g)|$.

We say that a group G *acts on* Δ if a homomorphism $\varphi : G \rightarrow \mathrm{Sym}(\Delta)$ is given (by specifying the image of a generator set of G). This action is *faithful* if its kernel $\ker(\varphi)$ is the identity. The image $\varphi(G) \leq \mathrm{Sym}(\Delta)$ is also denoted by G^Δ. In the special case when $G \leq \mathrm{Sym}(\Omega)$, $\Delta \subseteq \Omega$ is fixed by G, and φ is the restriction of permutations to Δ, we also denote G^Δ by $G|_\Delta$. The *orbit* of $\omega \in \Omega$ under $G \leq \mathrm{Sym}(\Omega)$ is the set of images $\omega^G := \{\omega^g \mid g \in G\}$. For $\Delta \subseteq \Omega$ and $g \in \mathrm{Sym}(\Omega)$, $\Delta^g := \{\delta^g \mid \delta \in \Delta\}$. A group $G \leq \mathrm{Sym}(\Omega)$ is *transitive* on Ω if it has only one orbit, and G is *t-transitive* if the action of G induced on the set of ordered t-tuples of distinct elements of Ω is transitive ($t \leq n$). The maximum such t is the *degree of transitivity* of G.

If $G \leq \mathrm{Sym}(\Omega)$ is transitive and $\Delta \subseteq \Omega$, then Δ is called a *block of imprimitivity* for G if for all $g \in G$ either $\Delta^g = \Delta$ or $\Delta^g \cap \Delta = \emptyset$. The group G is called *primitive* if all blocks have 0, 1, or $|\Omega|$ elements. If Δ is a block then the set of images of Δ is a partition of Ω, which is called a *block system*, and an action of G is induced on the block system. A block is called *minimal* if it has more than one element and its proper subsets of size at least two are not blocks. A block is called *maximal* if the only block properly containing it is Ω. A block system is *maximal* if it consists of minimal blocks, whereas a block system is *minimal* if it consists of maximal blocks. The action of G on a minimal block system is primitive.

For $\Delta \subseteq \Omega$ and $G \leq \mathrm{Sym}(\Omega)$, $G_{(\Delta)}$ denotes the *pointwise stabilizer* of Δ, namely, $G_{(\Delta)} = \{g \in G \mid (\forall \delta \in \Delta)(\delta^g = \delta)\}$. If Δ has only one or two elements, we often drop the set braces and parentheses from the notation; in particular, G_δ denotes the stabilizer of $\delta \in \Omega$. The *setwise stabilizer* of Δ is denoted by G_Δ (i.e., $G_\Delta = \{g \in G \mid \Delta^g = \Delta\}$). If $\Delta = (\delta_1, \ldots, \delta_m)$ is a *sequence* of elements of Ω then G_Δ denotes the pointwise stabilizer of that sequence (i.e., $G_\Delta = G_{(\{\delta_1,\ldots,\delta_m\})}$).

A group $G \leq \mathrm{Sym}(\Omega)$ is *semiregular* if $G_\delta = 1$ for all $\delta \in \Omega$, whereas G is *regular* if it is transitive and semiregular. A *Frobenius group* is a transitive group $G \leq \mathrm{Sym}(\Omega)$ that is not regular but for which $G_{\alpha\beta} = 1$ for all distinct $\alpha, \beta \in \Omega$.

If $g \in \mathrm{Sym}(\Omega)$ then a bijection $\varphi : \Omega \to \Delta$ naturally defines a permutation $\bar{\varphi}(g) \in \mathrm{Sym}(\Delta)$ by the rule $\varphi(\omega)^{\bar{\varphi}(g)} := \varphi(\omega^g)$ for all $\omega \in \Omega$. We say that $G \leq \mathrm{Sym}(\Omega)$ and $H \leq \mathrm{Sym}(\Delta)$ are *permutation isomorphic*, $H \sim G$, if there is a bijection $\varphi \colon \Omega \to \Delta$ such that $\bar{\varphi}(G) := \{\bar{\varphi}(g) \mid g \in G\} = H$.

Let G be an arbitrary group and let $H \leq S_k$ be a transitive permutation group. The *wreath product* $G \wr H$ consists of the sequences $(g_1, \ldots, g_k; h)$ where $g_i \in G$ for $i = 1, \ldots, k$ and $h \in H$. The product of $(g_1, \ldots, g_k; h)$ and $(\bar{g}_1, \ldots, \bar{g}_k; \bar{h})$ is defined as $(g_1 \bar{g}_{1^h} \ldots, g_k \bar{g}_{k^h}; h\bar{h})$.

1.2.3. Algorithmic Concepts

Groups in algorithms will always be input and output by specifying a list of generators.

Given $G = \langle S \rangle$, a *straight-line program of length m* reaching some $g \in G$ is a sequence of expressions $(w_1, \ldots, w_m)$ such that, for each i, w_i is a symbol for some element of S, or $w_i = (w_j, -1)$ for some $j < i$, or $w_i = (w_j, w_k)$ for some $j, k < i$, such that if the expressions are evaluated in G the obvious way then the value of w_m is g. Namely, the evaluated value of a symbol for a generator is the generator itself; the evaluated value of $w_i = (w_j, -1)$ is the inverse of the evaluated value of w_j; and the evaluated value of $w_i = (w_j, w_k)$ is the product of the evaluated values of w_j and w_k. Hence a straight-line program is an encoding of a sequence of group elements $(g_1, \ldots, g_m)$ such that $g_m = g$ and for each i one of the following holds: $g_i \in S$, or $g_i = g_j^{-1}$ for some $j < i$, or $g_i = g_j g_k$ for some $j, k < i$. However, the more abstract definition as a sequence of expressions not only requires less memory but also enables us to construct a straight-line program in one representation of G and evaluate it in another, which is an important feature of some algorithms.

The symbols $\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{R}$ denote the set of integers, nonnegative integers, and real numbers, respectively. Let

$$\mathcal{F} := \{f \colon \mathbb{N} \to \mathbb{R} \mid (\exists n_0 \in \mathbb{N})(\forall n > n_0)(f(n) > 0)\}$$

(i.e., functions that take positive values with finitely many exceptions). For

$f \in \mathcal{F}$, we define

$$\begin{aligned} O(f) &:= \{t \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(t(n) < cf(n))\}, \\ \Omega(f) &:= \{t \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(t(n) > cf(n))\}, \\ o(f) &:= \{t \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n > n_0)(t(n) < cf(n))\}, \end{aligned}$$

and

$$\Theta(f) := O(f) \cap \Omega(f).$$

Stated less formally, $t \in O(f)$ means that for large enough $n, t(n)/f(n)$ is bounded from above by an absolute constant c; $t \in \Omega(f)$ means that for large enough n, $t(n)/f(n)$ is bounded from below by an absolute positive constant c; and $t \in o(f)$ means that the limit of $t(n)/f(n)$ is 0.

For $t \in O(f)$, we also say that t is $O(f)$, and we use similar statements for Ω and Θ as well. We believe that this notation is more correct than the traditional $t = O(f)$ (see [Brassard and Bratley, 1988, Chap. 2] for the different variants of these notations).

We also introduce a "soft version" of the big-O notation. We write $t \in O^{\sim}(f)$ if $t(n) \leq Cf(n) \log^c n$ for large enough n (where c, C are positive constants). Logarithms are always of base 2.

1.2.4. Graphs

Let V be a set and $\mathcal{E}$ a subset of the two-element subsets of V. The pair $(V, \mathcal{E})$ is called a *graph* $\mathcal{X}(V, \mathcal{E})$. The elements of V and $\mathcal{E}$ are the *vertices* and the *edges* of $\mathcal{X}$, respectively. For $v \in V$, the number of edges containing v is the *degree* or *valency* of v, which we shall denote by $\deg(v)$. A graph $\mathcal{X}$ is called *regular* if all vertices have the same valency. We also say that an edge $\{u, v\} \in \mathcal{E}$ connects u and v. The set $N(v) := \{u \in V \mid \{u, v\} \in \mathcal{E}\}$ is the *neighborhood* of v in $\mathcal{X}$. A graph $\mathcal{X}$ is called *bipartite* if V can be partitioned into two sets A, B so that all edges of $\mathcal{X}$ connect some vertex in A with some vertex in B. The *automorphism group* $\text{Aut}(\mathcal{X})$ consists of those permutations of V that leave $\mathcal{E}$ invariant.

These notions can be generalized by requiring only that the elements of $\mathcal{E}$ are subsets of V, but of arbitrary size. Then $\mathcal{X}$ is called a *hypergraph*. A hypergraph $\mathcal{X}$ is *uniform* if all elements of $\mathcal{E}$ are of the same size.

Another generalization is when $\mathcal{E}$ consists of ordered pairs of V. Then $\mathcal{X}$ is called a *directed graph*. If we want to emphasize that a graph is directed then we shall use arrows above $\mathcal{E}$ and above the ordered pairs in $\mathcal{E}$. The *out-degree* of a vertex u is the number of edges $\overrightarrow{(u, v)}$ in $\mathcal{E}$. For a directed graph $\mathcal{X}(V, \vec{\mathcal{E}})$, the *underlying graph* of $\mathcal{X}$ is the graph $\mathcal{U}(V, E)$ with edge set $E = \{\{u, v\} \mid \overrightarrow{(u, v)} \in \vec{\mathcal{E}}\}$.

Let $\mathcal{X}(V, \mathcal{E})$ be a graph. A *walk* in $\mathcal{X}$ is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $\{v_i, v_{i+1}\} \in \mathcal{E}$ for all $i \in [0, k-1]$. A *path* is a walk with $v_i \neq v_j$ for all i, j with $0 \leq i < j \leq k$; a *cycle* is a walk with $v_0 = v_k$ and $v_i \neq v_j$ for all i, j with $0 \leq i < j \leq k-1$. We can define a binary relation $\mathcal{R}$ on V by letting $(u, v) \in \mathcal{R}$ if and only if there is a walk $(u = v_0, v_1, \ldots, v_k = v)$. Then $\mathcal{R}$ is an equivalence relation; the equivalence classes are called the *components* of $\mathcal{X}$. The graph $\mathcal{X}$ is *connected* if it has only one component.

If $\mathcal{X}$ is a directed graph then walks, cycles, and paths are defined similarly, but the binary relation $\mathcal{R}$ is not necessarily an equivalence relation. We may define another binary relation $\hat{\mathcal{R}}$ on V: $(u, v) \in \hat{\mathcal{R}}$ if and only if $(u, v) \in \mathcal{R}$ and $(v, u) \in \mathcal{R}$. Then $\hat{\mathcal{R}}$ is an equivalence relation, and its equivalence classes are called the *strongly connected components* of $\mathcal{X}$. The directed graph $\mathcal{X}$ is *strongly connected* if it has only one strongly connected component.

Cycle-free graphs are called *forests* and connected, cycle-free graphs are called *trees*. A *rooted tree* is a tree with a distinguished vertex. If $\mathcal{X}(V, \mathcal{E})$ is a rooted tree with root r then the *parent* of $v \in V \backslash \{r\}$ is the first vertex after v on the unique path from v to r. The *children* of $v \in V$ are those vertices whose parent is v. A vertex without children is called a *leaf*.

Let G be a group and $S \subseteq G$. The *Cayley graph* $\Gamma(G, S)$ is defined to have vertex set G; for $g, h \in G$, $\{g, h\}$ is an edge if and only if $gs = h$ for some $s \in S \cup S^{-1}$. The Cayley graph $\Gamma(G, S)$ is connected if and only if $G = \langle S \rangle$. Cayley graphs are regular and vertex-transitive (i.e., $\mathrm{Aut}(\Gamma(G, S))$ is a transitive subgroup of $\mathrm{Sym}(G)$).

Sequences will be denoted by enclosing their elements in parentheses. For a sequence L, $L[i]$ denotes the ith element of L.

The most abused notation is (a, b) for integers a and b. Depending on the context, it may mean a sequence with elements a and b, the set of real numbers between a and b, the set of integers between a and b (although, in this case, we shall prefer to use the closed interval $[a + 1, b - 1]$), or the permutation exchanging a and b.

1.3. Classification of Randomized Algorithms

As we shall see in numerous examples in this book, randomization can speed up many algorithms handling permutation groups. In other parts of computational group theory, randomization is even more important: For example, when dealing with matrix groups, the scope of efficient deterministic algorithms is very limited, both in the practical and theoretical sense of efficiency. Randomization also seems to be an indispensable tool in algorithms for black-box groups.

In this section, we describe the different types of randomized algorithms. Our discussion follows [Babai, 1997].

Computational tasks can be described by a relation $R(x, y)$ between an input string x and output string y. The relation $R(x, y)$ holds if y is a correct output for the task described by x. In group algorithms, the input usually contains a set of generators for a group. The output may consist of group elements, generating a desired subgroup of the input, but other types of output are also conceivable: The output could be a number (for example, the order of the input group) or a statement that group elements with the desired property do not exist (for example, when asking for a group element conjugating one given element to another one).

Note that there may be different correct outputs for the same input. We call a computational task *functional* if it has exactly one correct output for all inputs. For example, finding generators for a Sylow 2-subgroup is not a functional computational task, whereas finding the order of a group is. A special category of (functional) computational tasks is the class of *decision problems*. Here, the answer is a single bit, representing "yes" or "no." For example, determining solvability of the input group is a decision problem.

A (correct) *deterministic algorithm* computes an output $f(x)$ for all inputs x, so that $R(x, f(x))$ holds. A *randomized algorithm* uses a string r of random bits ("coin flippings") and returns the output $f(x, r)$. The output may not be correct for every sequence r.

We call a randomized algorithm *Monte Carlo* if for all inputs x,

$$\text{Prob}(R(x, f(x, r)) \text{ holds}) \geq 1 - \varepsilon,$$

where the value for the error term $\varepsilon < 1/2$ is specified in the description of the Monte Carlo algorithm. In most practical situations, the reliability of the algorithms can be improved by repeated applications, or by running the algorithms longer. Although we cannot formulate a theorem in the general setting considered in this section, at least the following holds for all Monte Carlo algorithms for permutation groups described in this book: The probability of an incorrect answer can be bounded from above by an arbitrary $\varepsilon > 0$, prescribed by the user. The associated cost is the running time of the algorithm multiplied by a factor $O(\log(1/\varepsilon))$.

A situation we can analyze here is the case of decision problems. Suppose that we have a Monte Carlo algorithm for a decision problem, with error probability $\varepsilon < 1/2$. Running the algorithm t times, and taking a majority vote of the results, we can increase the reliability to at least $1 - \delta^t$, with $\delta := 2\sqrt{\varepsilon(1-\varepsilon)} < 1$: The probability of error is at most

$$\sum_{k=t/2}^{t} \binom{t}{k} \varepsilon^k (1-\varepsilon)^{t-k} < (1-\varepsilon)^t \sum_{k=t/2}^{t} \binom{t}{k} \left(\frac{\varepsilon}{1-\varepsilon}\right)^{t/2} < \delta^t.$$

Babai also discusses *one-sided Monte Carlo algorithms* (1MC algorithms) for decision problems. These are algorithms where at least one of the outputs is guaranteed to be correct: If the correct answer is "yes," a 1MC algorithm may err; if the correct answer is "no," the algorithm must always output "no." Hence, an output "yes" is always correct, whereas an output "no" may not be. The *co-1MC algorithms* are defined by exchanging the words "yes" and "no" in the definition of 1MC algorithms.

In the context of group theoretical algorithms, the notion of 1MC algorithms can be extended to most computational tasks, since the error is usually one-sided: For example, when computing generators U for the normal closure of some $H \leq G$ (cf. Section 2.3.3), we never place an element of G on the generator list U that is not in $\langle H^G \rangle$. The error we may commit is that $\langle U \rangle$ is a proper subgroup of $\langle H^G \rangle$. Another example is the Monte Carlo order computation in permutation groups (cf. Section 4.5): The result we obtain is always a lower bound for $|G|$, because we do not place permutations not belonging to G into the strong generating set of G.

An important subclass of Monte Carlo algorithms is the class of *Las Vegas algorithms*. This term was introduced in [Babai, 1979] to denote Monte Carlo algorithms that never give an incorrect answer. The output is either correct (with the prescribed probability at least $1 - \varepsilon$) or the algorithm reports failure. Here, ε may be any given constant less than 1, since the probability of an (always correct) output can be increased to at least $1 - \varepsilon^t$ by running the algorithm t times.

Las Vegas algorithms are preferable over general Monte Carlo algorithms for many reasons. One of them is the certainty of the answer. Another one is that recognizing the correct answer often allows early termination. Finally, if we can guarantee that the output is always correct then we may use heuristics to speed up the algorithm, even in cases when we cannot estimate the probability of error for our heuristics.

In practice, if a Las Vegas algorithm reports failure then we rerun it until the correct output is produced. The definition of Las Vegas algorithms can be reformulated so that they always return a correct answer, with the running time estimate replaced by an estimate of the *expected* running time.

A Monte Carlo algorithm, combined with a deterministic checking of the result, becomes a Las Vegas algorithm, because we can recognize that the Monte

Carlo algorithm returned an incorrect output, and report failure. Another way to obtain Las Vegas algorithms is by using both 1MC and co-1MC algorithms for a decision problem. Running both of these algorithms, with probability at least $1 - \varepsilon$ one of them will return a guaranteed correct output. If both algorithms return an answer that may not be correct, then we report failure. An important direction of current research is the upgrading of Monte Carlo algorithms for permutation groups to Las Vegas type. We shall describe a result in this direction in Section 8.3.