データ構造とアルゴリズム分析

エディション3.2（Java版）

クリフォード・A・シェ

イファー コンピュータサイ

エンス学科 バージニア工科大

学

ブラックスバーグ, VA 24061

2013年3月28日

アップデート 3.2.0.10

変更点のリストについては

~ http://people.cs.vt.edu/shaffer/Book/errata.html

ドーバー出版

(http://store.doverpublications.com/0486485811.html)。

このテキストに関する詳細は、以下を参照のこと。

~ http://people.cs.vt.edu/shaffer/Book/.

内容

序文

私たちがデータ構造を学ぶのは、より効率的なプログラムを書けるようになるためだ。しかし、毎年新しいコンピューターが高速化しているのに、なぜプログラムは効率的でなければならないのだろうか? その理由は、能力とともに私たちの野心も成長するからである。コンピュータのパワーとストレージ能力における現代の革命は、効率化の必要性を時代遅れにするのではなく、より複雑なタスクに挑戦する際に効率化の必要性を高めるだけなのだ。

プログラムの効率性を追求することは、健全な設計や明瞭なコーディングと対立する必要はないし、対立すべきではない。効率的なプログラムの作成は、「プログラム・トリック」とはほとんど関係がなく、むしろ情報の優れた構成と優れたアルゴリズムに基づいている。明確な設計の基本原則を習得していないプログラマは、効率的なプログラムを書くことはできないだろう。逆に、開発コストや保守性に関する懸念は、非効率的なパフォーマンスを正当化するための言い訳に使うべきではない。設計の汎用性は、性能を犠牲にすることなく達成することが可能であり、また達成すべきであるが、これは設計者が性能測定の方法を理解し、設計と実装のプロセスの不可欠な一部としてそれを行う場合にのみ可能である。ほとんどのコンピュータ・サイエンスのカリキュラムは、優れたプログラミング・スキルは、基本的なソフトウェア工学の原則に重点を置くことから始まると認識している。そして、プログラマーが明確なプログラム設計と実装の原則を学んだら、次のステップは、データ構

成とアルゴリズムがプログラムの効率に及ぼす影響を研究することである。

アプローチ本書はデータを表現するための多くの技法について述べている。これらのテクニックは、以下の原則の中で紹介されている：

1. 各データ構造と各アルゴリズムにはコストとベネフィットがある。新たな設計課題に対応するためには、コストとベネフィットの評価方法を十分に理解する必要がある。そのためには、アルゴリズム解析の原理を理解し、さらに採用する物理的媒体（例えば、ディスクに保存されるデータとメイン・メモリに保存されるデータ）の重大な影響を理解する必要がある。
2. コストとベネフィットに関連して、トレードオフという概念がある。例えば、必要なスペースを増やす代わりに必要な時間を減らす、あるいはその逆はよくあることだ。プログラマーはあらゆる場面でトレードオフの問題に直面する。

ソフトウェア設計と実装の各段階において、そのコンセプトは深く浸透していなければならない。

3. プログラマーは、車輪の再発を避けるために、一般的な慣習について十分に知っておく必要がある。したがって、プログラマーはよく使われるデータ構造とそれに関連するアルゴリズム、そしてプログラミングで最も頻繁に遭遇するデザインパターンを学ぶ必要がある。
4. データ構造はニーズに従う。プログラマーは、まずアプリケーションのニーズを評価し、それに見合った機能を持つデータ構造を見つけることを学ばなければならない。そのためには、原則1、2、3の能力が必要である。

長年にわたってデータ構造を教えてきて、設計の問題が私のコースで果たす役割がますます大きくなっていることに気づいた。このことは、この教科書の様々な版を通して、デザイン・パターンとジェネリック・インターフェースのカバー範囲が広がっていることからもわかる。第1版では、デザイン・パターンについての言及はなかった。第2版では、いくつかのパターンの例を限定的に取り上げ、ADTという辞書を導入した。第3版では、この本で扱われている基本的なデータ構造とアルゴリズムをプログラミングする際に遭遇する、いくつかのデザイン・パターンを明確に取り上げている。

授業で本を使うデータ構造とアルゴリズムの教科書は、教本か百科事典の2つに分類される傾向がある。両方を行おうとする本は、たいてい両方とも失敗する。本書は教育用テキストとして意図されている。実務家にとっては、教科書の実装をたくさん暗記することよりも、ある問題を最適に解決するデータ構造を選択したり設計したりするために必要な原理を理解することの方が重要だと私は考えている。したがって、このテキストは、すべてではありませんが、ほとんどの標準的なデータ構造をカ

バーする教材として設計しました。重要な原則を説明するために、広く採用されていないデータ構造もいくつか含まれている。また、将来広く使われるようになるであろう比較的新しいデータ構造も含まれている。

本書は、学部課程において、上級下級（2年生または3年生レベル）のデータ構造コース、または上級レベルのアルゴリズムコースで使用することを目的としている。第3版では、アルゴリズムコースでの使用をサポートするために、新しい資料が追加された。通常、このテキストは、データ構造の最初の導入となることが多い標準的な1年生レベルの「CS2」コースを超えるコースで使用される。本書の読者は、少なくともJavaに多少触れるなど、通常2学期相当のプログラミング経験が必要である。また、すでに再帰に慣れ親しんでいる読者は有利である。また、データ構造を学ぶ学生は、離散数学の良いコースを最初に修了していることが有益である。それにもかかわらず、第2章では、 本書での使用を理解するのに必要なレベルで、前提条件となる数学的トピックをある程度完全に調査しようと試みている。読者は、馴染みのない数学的トピックに遭遇したとき、必要に応じて適切なセクションを参照することができる。

2年生レベルのクラスで、基本的なデータ構造や解析の知識が少ししかない（つまり、従来のCS2コースと同等の知識しかない）学生であれば、第1章から第11章を詳細にカバーし、第13章から特定のトピックを取り上げるかもしれない。私の2年生レベルのクラスでは、この本をそのように使っている。それ以上のバックグラウンドを持つ学生は、第1章をカバーし、第2章は参考文献を除いてほとんど飛ばし、第3章と第4章を簡単にカバーし、それから第5章から第12章を詳しくカバーするかもしれない。この場合も、講師が選ぶプログラミングの課題によっては、第13章から特定のトピックだけを取り上げるかもしれない。上級レベルのアルゴリズムコースでは、第11章と第14章から第17章に焦点を当てる。

第13章は、より大規模なプログラミング演習のための資料として意図されている。私は、データ構造コースを受講するすべての学生に、何らかの高度な木構造、あるいは第12章のスキップリストや疎行列表現など、同等の難易度を持つ別の動的構造を実装することを要求することを推奨する。これらのデータ構造はいずれも、バイナリ探索木よりも実装が著しく難しいということはなく、第5章を修了した後であれば、どのデータ構造も学生の能力の範囲内にあるはずである。

私は納得のいく順序で説明するよう努めたが、指導者は自由にトピックを並べ替えることができる。本書は、読者が第1章から第6章までをマスターすれば、残りの内容は比較的依存関係が少なくなるように書かれている。明らかに、外部ソートは内部ソートとディスクファイルの理解に依存している。UNION/FINDアルゴリズムに関する6.2節は、Kruskalの最小コストスパニングツリーアルゴリズムで使用されている。自己組織化リストに関する9.2節では、8.3節で取り上げたバッファ置換スキームに言及している。第14章では、本書全体を通しての例を用いている。セクション17.2はグラフの知識に依存している。それ以外では、ほとんどのトピックは、同じ章内の以前のトピックに依存している。

ほとんどの章の最後には、"Further Reading "と題されたセクションが

ある。これらのセクションは、紹介したトピックに関する参考文献の包括的なリストではない。むしろ、読者にとって特別に有益であり、また楽しいと思われる書籍や論文を、私の意見として紹介している。場合によっては、教養あるコンピュータ科学者であれば誰でも知っているような著作を紹介することもある。

Javaの使用： プログラミングの例はJavaで書かれているが、Javaに馴染みのない人が本書を読むことを妨げるつもりはない。Javaの長所を生かしつつ、できるだけわかりやすい例題になるように努めた。ここでは、Javaはあくまでもデータ構造の概念を説明するためのツールとして使用している。特に、クラス、プライベート・クラス・メンバ、インターフェイスなど、実装の詳細を隠すJavaの機能を活用しています。言語のこれらの機能は、抽象データ型に具現化される論理設計と、データ構造に具現化される物理的実装を分離するという重要な概念をサポートしている。

他のプログラミング言語と同様、Javaにも利点と欠点の両方がある。Javaは小さな言語である。通常、何かをするための言語機能は1つしかなく、これが正しく使われれば、プログラマーを明瞭な方向に向かわせるという嬉しい傾向がある。この点で、JavaはCやC++よりも優れている。Javaは、リストやツリーといった伝統的なデータ構造のほとんどを定義し、使用するのに適している。一方、Javaはファイル処理に使うと、面倒で効率が悪くなる。また、メモリの細かい制御が必要な場合にも、Javaは不向きな言語である。たとえば、セクション12.3で説明するようなメモリ管理を必要とするアプリケーションは、Javaで書くのは難しい。この文章では一つの言語にこだわりたいので、他のプログラマーと同じように、良い点とともに悪い点も受け止めなければならない。最も重要な問題は、その考えが特定の言説言語にとって自然なものであるかどうかにかかわらず、考えを伝えることである。ほとんどのプログラマーはキャリアを通じて様々なプログラミング言語を使うだろうし、本書で説明されているコンセプトは様々な状況で役に立つはずだ。

オブジェクト指向プログラミングの重要な機能である継承は、コード例では控えめに使われている。継承は、プログラマーが重複を避け、バグを最小限に抑えるための重要なツールである。しかし、教育的な観点からは、継承は1つの論理ユニットの説明を複数のクラスに分散させる傾向があるため、コード例を理解しにくくすることが多い。従って、私のクラス定義では、継承が明示的に説明するポイントに関連する場合にのみ継承を使用しています（例えば、セクション5.3.1）。これは、プログラマーが同じようにすべきことを意味するものではありません。コードの重複を避け、エラーを最小限にすることは重要な目標です。プログラミングの例は、データ構造の原則を説明するものとして扱ってください。

ひとつ苦渋の決断を迫られたのは、コード例でジェネリックスを使うかどうかだった。本書の初版ではジェネリックスは使われていなかった

。しかし、あれから数年、Javaは成熟し、コンピューター・サイエンスのカリキュラムでの使用も大幅に拡大した。現在では、本書の読者がジェネリック構文に精通していることを前提としている。そのため、コード例ではジェネリックを多用しています。

私の実装は、データ構造の原則を具体的に示すものであり、文章による説明を補助するものである。コード例は、関連するテキストと切り離して読んだり使ったりすべきではない。なぜなら、各実験例のドキュメントの大部分は、コードではなくテキストに含まれているからだ。コードはテキストを補足するものであり、その逆ではありません。これらは、商業的な品質のクラス実装のシリーズではありません。標準的なデータ構造の完全な実装を自分のコードで使いたいのであれば、インターネットで検索するのがよいでしょう。

例えば、コード例では、健全なプログラミングの実践よりも少ないパラメータ・チェックしか提供していません。いくつかのパラメータ・チェックと他の制約（例えば、空のコンテナから値が削除されるかどうか）のテストは

Assertクラスのメソッド呼び出しの形式。メソッド**Assert.**notFalseは論理式を取ります。この式が**false**と評価された場合、メッセージが出力され、プログラムは直ちに終了します。メソッド**Assert.**notNullはクラス**Object**への参照を取り、 参照の値が**NULL**の場合、プログラムを終了します。(正確には、**IllegalArgument- Exception**をスローし、プログラマが例外を処理するアクションを取らない限り、プログラムは終了する)。関数が不正なパラメータを受け取ったときにプログラムを終了させることは、実際のプログラムでは一般的に望ましくないと考えられているが、データ構造がどのように動作するかを理解するには十分である。実際のプログラミング・アプリケーションでは、Javaの例外処理機能を使用して入力データ・エラーに対処すべきである。しかし、アサーションは、データ構造がどのように動作するかを明確にするのに十分であり、真の例外処理に簡単に変更できる方法で、必要な条件を指示するための、より単純なメカニズムを提供します。

私は本文中で、"Java実装 "と "pseu-docode "を区別している。Java実装と書かれたコードは、実際に1つ以上のJavaコンパイラでコンパイルされ、テストされている。擬似コードの例は、Javaの構文に忠実であることが多いが、通常、1行以上の高レベルの記述が含まれている。擬似コードは、より単純だが正確さに欠ける記述の方が、教育的な利点が大きいと思われる場合に使用される。

演習とプロジェクトデータ構造の適切な実装と分析は、本を読むだけでは学べません。実際のプログラムを実装し、常にさまざまなテクニックを比較しながら、ある状況において何が本当に最適なのかを確認しながら練習する必要があります。

データ構造のコースの最も重要な側面のひとつは、リンクリストやツリーなどのデータ構造を実装することで、ポインタやダイナミック・メモリ・アル・ロケーションを使ったプログラミングを実際に学ぶ場であるということだ。多くの場合、再帰を学ぶ場でもある。私たちのカリキ

ュラムでは、このコースは学生が重要な設計を行う最初のコースです。なぜなら、このコースでは重要な設計演習を行うために実際のデータ構造が必要になることが多いからです。最後に、メモリベースとディスクベースのデータアクセスの基本的な違いは、実践的なプログラミングの経験なしには理解できない。これらの理由から、データ構造コースは重要なプログラミングの要素なしには成功しない。本学科では、データ構造コースはカリキュラムの中で最も難しいプログラミングコースの一つである。また、学生は分析能力を養うために問題に取り組む必要がある。私は450以上の練習問題とプログラミング・プロジェクトの提案を提供する。私は読者に次のことを勧める。それを利用するためだ。

著者への連絡と補足資料：このような本には間違いや改善の余地が必ずあります。バグ報告や建設的な批評を歓迎する。電子メールでのご連絡は、インターネット経由でshaffer@vt.edu。また、コメントは下記まで郵送してください。

クリフ・シェイファー
バージニア工科大学コンピュータ
ーサイエンス学部
ブラックスバーグ, VA 24061

本書の電子版と、授業で使用する講義ノートは、以下のサイトで入手できる。

~ http://www.cs.vt.edu/shaffer/book.html.

この本で使われているコード例は、同じサイトで入手できる。ヴァージニア工科大学の2年生レベルのデータ構造クラスのオンライン・ウェブ・ページは以下にある。

~ cs3114http://courses.cs.vt.edu/.

この教科書の読者は、我々のオープンソースのオンライン電子教科書プロジェクト、OpenDSA **(http://algoviz.org/Op**enDSA)に興味を持つだろう。OpenDSAプロジェクトのゴールは、教科書に載っているような質の高いコンテンツと、すべてのアルゴリズムとデータ構造に関するアルゴリズムの視覚化、そして豊富なインタラクティブな練習問題を組み合わせたチュートリアルの完全なコレクションを作ることです。完成すれば、OpenDSAはこの本を再出版する予定である。

本書は著者がLATEXを用いて組版した。書誌はBIBTEXで作成。索引は**makeindexで**作成した。図はほとんど**Xfigで**描いた。図3.1と図9.10は部分的にMath- ematicaで作成した。

謝辞一冊の本を作るには、多くの人々の協力が必要である。本書の出版にご協力いただいた方々の一部をここに記しておきたい。どうしても漏れてしまうことをお詫びする。

ヴァージニア工科大学は、1994年秋のサバティカル休暇を通じてこのプロジェクト全体を可能にしてくれた。本書の様々な版を執筆してきた時期の私の部門長、デニス・カフラとジャック・キャロルは、このプロジェクトに揺るぎない精神的支援を与えてくれた。マイク・キーナン、レ

ニー・ヒース、ジェフ・シェイファーには、各章の初期版について貴重な意見をいただいた。またレニー・ヒースには、長年にわたってアルゴリズムと解析（そしてその両方を学生に教える方法）について刺激的な議論を交わしてくれたことに感謝したい。スティーブ・エドワーズには、第2版と第3版のC++とJavaのコード・バージョンの様々な再設計に多くの時間を費やしてくれたこと、またプログラム設計の原理について多くの時間を議論してくれたことに、特に感謝したい。Mathematica を手伝ってくれたLayne Watson，多くの技術的援助をくれたBo Begole，Philip Isenhour，Jeff Nielsen，Craig Strubleに感謝する．また，C++とJavaに関するたくさんのくだらない質問に答えてくれたBill Mc- Quain，Mark Abrams，Dennis Kafuraに感謝します．

このマニュスクリプトの様々な版について、多くの査読者の方々に心から感謝している。初版の査読者には、J. David Bezek（University of America）が含まれる。

Evansville）、Douglas Campbell （Brigham Young University）、Karen Davis （University of Cincinnati）、Vijay Kumar Garg （University of Texas - Austin）、Jim Miller （University of Kansas）、Bruce Maxim （University of Michigan - Dearborn）、Jeff Parker （Agile Networks/Harvard）、Dana Richards （George Mason University）、Jack Tan （University of Houston）、Lixin Tao （Concordia University）。彼らの協力がなければ、本書にはもっと多くの技術的な誤りと、もっと少ない洞察が含まれていただろう。

第2版については、以下の査読者に感謝したい：Gurdip Singh（カンザス州立大学）、Peter Allen（コロンビア大学）、Robin Hill（ワイオミング大学）、Norman Jacobson（カリフォルニア大学アーバイン校）、Ben Keller（ミシガン東部大学）、Ken Bosworth（アイダホ州立大学）。さらに、ニール・スチュワートとフランク・J・テッセンのコメントと改良のためのアイデアに感謝したい。

第3版の査読者には、Randall Lechlitner（University of Houstin, Clear Lake）とBrian C. Hipp（York Technical College）が含まれている。彼らのコメントに感謝する。

プレンティスホールは、初版と第2版の印刷出版社である。そこでの多くの人々の努力なしには、このようなことは不可能であった。オーサーは自分たちだけでは印刷に耐えうる本を作ることはできない。ケイト・ハーゲット、ペトラ・レクター、ローラ・スティール、アラン・アプトには、長年にわたって編集に携わってもらった。第2版ではアーウィン・ザッカー、オリジナルのC++版ではキャスリーン・カレン、Java版ではエド・デフェリッピスというプロダクション・エディターが、最後のひどいラッシュの中、すべてをスムーズに進めてくれた。ビル・ゾブリストとブルース・グレゴリー（だと思う）には、最初に私をこの世界に引き込んでくれたことに感謝している。その他、私を助けてくれたPrentice HallのTruly Donovan、Linda Behrens、Phyllis Bregmanに感謝する。トレイシー・ダンケルバーガーには、コピーライトの返却を手伝ってもらった。

プレンティス・ホールの他の多くの方々にも、私の知らないところでお世話になった。

この第3版の印刷出版を引き受けてくれたドーバー出版社のシェリー・クロンゼックに感謝している。Java版とC++版の両方を増補し、多くの矛盾点を修正したこの第3版は、これまでで最高のものだと確信している。しかし、学生たちが無料のオンライン教科書を好むのか、それとも低コストの印刷製本版を好むのか、私たちの誰も本当のところはわからない。最終的には、2つの形式がより多くの選択肢を提供することで、相互支援になると信じています。プロダクション・エディターのジェームス・ミラーとデザイン・マネージャーのマリー・ザツキェヴィッチは、最高の品質で制作されるよう努力してきた。

データ構造について教えてくれたハナン・サメットに感謝の意を表したい。ここで紹介する哲学の多くも彼から学んだものである。妻のテリーには愛とサポートを、娘のイレーナとケイトには働き過ぎを楽しく紛らわしてくれたことに感謝する。最後に、そして最も重要なことだが、何が重要で何が重要でないかを教えてくれた、長年にわたってデータ構造を学んできた学生たち全員に感謝する。

データ構造コースではスキップすべきであり、彼らが提供した多くの新しい洞察である。本書は彼らに捧げられる。

クリフ・シ
ェイファー ヴァー
ジニア州ブラック
スバーグ

第一部

予備知識

1

データ構造とアルゴリズム

テキサス州ダラスから500マイル以内に、人口25万人以上の都市はいくつありますか? 私の会社で年収10万ドル以上の人は何人いますか? 1,000マイル未満のケーブルで、すべての電話顧客を接続できるか? このような質問に答えるには、必要な情報を持っているだけでは不十分である。私たちのニーズを満たすために、答えをすぐに見つけられるように情報を整理しなければなりません。

情報を表現することはコンピュータ・サイエンスの基本である。ほとんどのコンピュータ・プログラムの主な目的は、計算を実行することではなく、情報を保存し検索することである。このため、データ構造とそれを操作するアルゴリズムの研究は、コンピュータ・サイエンスの中核をなすものである。本書は、効率的な処理をサポートするために情報をどのように構造化するかを理解する手助けをするものである。

本書の主な目的は3つある。第一は、よく使われるデータ構造を紹介することである。これらはプログラマーの基本的なデータ構造の "ツールキット "を形成する。多くの問題に対して、ツールキットの中のいくつかのデータ構造が良い解決策を提供してくれます。

第二の目標は、トレードオフの考え方を導入し、すべてのデータ構造にはコストとベネフィットがあるという概念を強化することである。これは、それぞれのデータ構造について、典型的な操作に必要なスペースと時間の量を説明することによって行われる。

第三の目標は、データ構造やアルゴリズムの有効性を測定する方法を教えることである。このような測定を通してのみ、自分のツールキットの中でどのデータ構造が新しい問題に最も適しているかを判断することができる。また、このテクニックを使うことで、あなたや他の人が発明するかもしれない新しいデータ構造の良し悪しを判断することができる。

問題解決には多くのアプローチがある。その中からどのように選択すればよいのだろうか? コンピュータ・プログラム設計の核心には、2つの（時には相反する）目標がある:

1. 理解しやすく、コーディングしやすく、デバッグしやすいアルゴリズムを設計すること。
2. コンピュータのリソースを効率的に使用するアルゴリズムを設計する。

理想的には、出来上がったプログラムはこれら両方の目標に忠実である。このようなプログラムは "エレガント "であると言えるかもしれない。本書で紹介するアルゴリズムやプログラムコードの例は、この意味でエレガントであることを試みているが、本書の目的はゴール(1)に関連する問題を明確に扱うことではない。これらは主にソフトウェア工学の分野での問題である。むしろ、本書はゴール(2)に関する問題を主に扱っている。

どうやって効率を測るのか？第3章では、**漸近解析と**呼ばれるアルゴリズムやコンピュータプログラムの効率を評価する方法について説明する。漸近解析では、問題の本質的な難易度を測ることもできる。残りの章では、漸近解析のテクニックを使って、紹介されているすべてのアルゴリズムの時間コストを推定します。これにより、各アルゴリズムが、同じ問題を解くための他のアルゴリズムと比較して、効率性の点でどのような違いがあるかを見ることができます。

この第1章では、データ構造の選択と使用に関するいくつかの高次の問題を提示することで、この後に続く問題の舞台を整える。まず、設計者が目の前のタスクに適したデータ構造を選択するプロセスを検討する。次に、プログラム設計における抽象化の役割について考える。デザイン・パターンの概念について簡単に説明し、いくつかの例を示します。本章の最後は、問題、アルゴリズム、プログラムの関係についての考察で終わる。

1.1 データの哲学 構造

1.1.1 データ 構造の必要性

コンピューターがますます高性能になるにつれ、プログラムの効率性はそれほど重要ではなくなってきていると思うかもしれない。結局のところ、プロセッサーのスピードとメモリーのサイズは依然として向上し続

けている。現在抱えているかもしれない効率性の問題は、明日のハードウェアによって解決されるのではないだろうか?

より強力なコンピューターが開発されるにつれ、私たちのこれまでの歴史は常に、より洗練されたユーザー・インターフェース、より大きな問題サイズ、あるいは以前は計算不可能とされていた新しい問題など、より複雑な問題に取り組むために、その追加された計算パワーを使ってきた。より複雑な問題は、より多くの計算を要求し、効率的なプログラムの必要性をさらに大きくする。さらに悪いことに、タスクが複雑になればなるほど、それは我々の日常的な経験とは似て非なるものになる。今日のコンピュータ科学者は、効率的なプログラム設計の背後にある原理を完全に理解するよう訓練されなければならない。

最も一般的な意味では、データ構造とはあらゆるデータ表現とそれに関連する操作のことである。コンピュータに格納されている整数や浮動小数点数でさえ、単純なデータ構造と見なすことができる。より一般的には、「データ構造」という用語は、データ項目の集まりの構成や構造化という意味で使われる。配列に格納された整数のソートされたリストは、そのような構造化の例である。

データ項目のコレクションを格納する十分なスペースがあれば、コレクション内の指定された項目を検索したり、データ項目を任意の順序で印刷またはその他の方法で処理したり、特定のデータ項目の値を変更したりすることは常に可能である。このように、どのようなデータ構造でも必要な操作はすべて可能である。しかし、適切なデータ構造を使うかどうかが、数秒で実行できるプログラムと何日もかかるプログラムとの違いになる。

必要な**リソース制約の**範囲内で問題を解決できる場合、その解は**効率的で**あると言われる。リソース制約の例としては、データを格納するために利用可能な総スペース（場合によっては、メインメモリとディスクのスペース制約に分割される）、および各サブタスクの実行に許容される時間がある。解が特定の要件を満たすかどうかに関係なく、既知の代替案よりも少ないリソースで済む場合、その解は効率的であると言われることがある。ソリューションの**コストは**、ソリューションが消費するリソースの量である。多くの場合、コストは時間などの1つの主要なリソースで測定され、そのソリューションは他のリソース制約を満たすという暗黙の前提がある。

人がプログラムを書くのは問題を解決するためであることは言うまでもない。しかし、特定の問題を解決するためにデータ構造を選択する際には、この真理を念頭に置くことが極めて重要である。まず問題を分析し、達成すべきパフォーマンス目標を決定することによってのみ、その仕事に適したデータ構造を選択する望みが持てる。下手なプログラム設計者は、この分析ステップを無視し、使い慣れているが問題には不適切なデータ構造を適用してしまう。その結果、通常遅いプログラムになる。逆に、より単純な設計で実装すれば性能目標を達成できるプログラムを「改善」するために複雑な表現を採用する意味はない。

問題を解決するためにデータ構造を選択するときは、以下のステップに従うべきである。

1. 問題を分析し、対応しなければならない基本操作を決定する。基本操作の例としては、データ構造へのデータ項目の挿入、データ構造からのデータ項目の削除、指定されたデータ項目の検索などがあります。
2. 各オペレーションのリソース制約を定量化する。
3. これらの要件を最もよく満たすデータ構造を選択する。

データ構造を選択するためのこの3段階のアプローチは、設計プロセスにおけるデータ中心の視点を運用するものである。最初の関心事はデータとそれに対して実行される操作であり、次の関心事はそれらのデータの表現であり、最後の関心事はその表現の実装である。

検索、データレコードの挿入、データレコードの削除など、特定の重要な操作に関するリソースの制約は、通常、データ構造の選択プロセスを推進する。これらの操作の相対的な重要性に関する多くの問題は、以下の3つの質問によって解決される：

- すべてのデータ項目は最初にデータ構造に挿入されるのか、それとも挿入は他の操作と一緒に行われるのか? 静的アプリケーション（データが最初にロードされ、変更されることがない）では、一般的に、動的アプリケーションよりも効率的な実装を得るために単純なデータ構造しか必要としない。
- データ項目は削除できるのか? もしそうなら、おそらく実装が複雑になるでしょう。
- すべてのデータ項目は、明確に定義された順序で処理されるのか、それとも特定のデータ項目の検索が許可されるのか。「ランダム・アクセス "検索は一般に、より複雑なデータ構造を必要とする。

1.1.2 コストと メリット

それぞれのデータ構造には、関連するコストとメリットがある。実際には、あるデータ構造がすべての状況において他のデータ構造より優れているということはほとんどない。もしあるデータ構造やアルゴリズムがすべての点で他のものより優れていれば、劣っている方のデータ構造はとっくに忘れ去られているのが普通です。本書で紹介するほぼすべてのデータ構造とアルゴリズムについて、それが最良の選択である例を紹介する。驚くような例もあるかもしれない。

データ構造には、格納するデータ項目ごとに一定のスペースが必要であり、1つの基本操作を実行するのに一定の時間が必要であり、プログラミングには一定の労力が必要である。それぞれの問題には、利用可能なスペースと時間に対する制約がある。問題に対する解はそれぞれ、基本操作を相対的な割合で使用するものであり、データ構造の選択プロセスはこれを考慮しなければなりません。問題の特徴を注意深く分析して初めて、そのタスクに最適なデータ構造を決定することができます。

例1.1 銀行は顧客との様々な取引をサポートしなければならない

が、ここでは顧客が口座の開設、口座の閉鎖、口座へのお金の追加や引き出しを希望する単純なモデルを検討する。(1)銀行が顧客とのやり取りで使用する物理的なインフラとワークフロー・プロセスに対する再要件、(2)口座を管理するデータベース・システムに対する要件。

一般的な顧客は、口座を開設したり閉鎖したりする頻度は、口座にアクセスする頻度よりはるかに少ない。顧客は、口座が作成されたり削除されたりする間、何分も待つことを厭わないが、入出金のような個々の口座取引のために短時間以上待つことを厭わないのが普通である。これらの観察は、問題の時間的制約に対する非公式な特定事項と考えることができる。

銀行は2段階のサービスを提供するのが一般的である。窓口または現金自動預け払い機（ATM）は、顧客のアクセスをサポートする。

口座の残高や入出金などの最新情報を提供する。通常、特別サービス担当者が（制限時間内に）口座の開設や解約に対応する。テラーやATMでの取引にはほとんど時間がかからない。口座の開設や解約にはかなり時間がかかる（顧客の立場からすると、おそらく1時間程度）。

データベースの観点からは、ATMトランザクションがデータベースを大きく変更することはないことがわかる。簡単のために、お金が追加されたり削除されたりした場合、このトランザクションは単に口座レコードに保存されている値を変更するだけであると仮定する。データベースに新しい口座を追加するには数分かかる。口座の削除には時間的な制約は必要ない。なぜなら、顧客から見て重要なのは、すべての資金が再回転されること（引き出しに相当する）だけだからである。銀行の観点からは、営業時間外や月次の口座サイクル終了時に、データベースシステムから口座記録が削除される可能性がある。

顧客口座を管理するデータベース・システムで使用するデータ構造の選択を考えるとき、削除のコストにはほとんど関心がないが、検索の効率が高く、挿入の効率が中程度のデータ構造が、この問題で課されるリソースの制約を満たすはずであることがわかる。レコードは一意の口座番号でアクセスできる（**完全一致クエリー**と呼ばれることもある）。これらの要求を満たすデータ構造の1つが、9.4章で説明したハッシュテーブルです。ハッシュ表は極めて高速な完全一致検索を可能にする。レコードのスペースに影響を与えない変更であれば、レコードを素早く変更することができる。ハッシュ表は新しいレコードの効率的な挿入もサポートする。削除も効率的にサポートできるが、削除が多すぎると残りの操作のパフォーマンスが多少低下する。しかし、ハッシュ・テーブルを定期的に再編成することで、システムを最高の効率に戻

すことができる。このような再編成は、ATMのトランザクションに影響を与えないようにオフラインで行うことができる。

例1.2 ある会社が、アメリカの市や町に関する情報を含むデータベースシステムを開発している。何千もの市や町があり、データベース・プログラムでは、ユーザが特定の場所に関する情報を名前で検索できるようにする必要があります（完全一致クエリのもう1つの例）。また、所在地や人口規模などの属性について、特定の値または値の範囲に一致するすべての場所を検索できるようにする必要があります。これは**範囲クエリとして**知られています。

合理的なデータベースシステムは、典型的なユーザーの忍耐力を満足させるのに十分な速さでクエリに答えなければならない。完全一致のクエリであれば、数秒で十分である。データベースが、クエリ仕様に一致する多くの都市を返すことができる範囲クエリをサポートすることを意図している場合、全体のオペレーションは、クエリ仕様に一致する多くの都市を返すことができる範囲クエリをサポートすることを意図している。

おそらく1分程度であろう。この要件を満たすには、個々の都市に対する一連の操作ではなく、範囲内のすべての都市をバッチとして処理することで、範囲クエリを効率的に処理する操作をサポートする必要がある。

前の例で提案したハッシュ・テーブルは、効率的な範囲クエリを実行できないため、都市データベースの実装には不適切である。セクション10.5.1のB^+-treeは、大規模なデータベース、データレコードの挿入と削除、範囲クエリをサポートしている。しかし、CDで配布される地図帳やウェブサイトからアクセスする地図帳のように、データベースが一度作成され、その後変更されることがない場合は、セクション10.1で説明したような単純な線形インデックスが適している。

1.2 抽象データ型とデータ 構造体

前節では、「データ項目」と「データ構造」という用語を定義せずに使用した。このセクションでは、用語の説明と、データ構造を選択するための3ステップ・アプローチで具現化される設計プロセスの動機付けを行う。この動機付けは、コンピュータ・プログラムの膨大な複雑さを管理する必要性から生じている。

型は値の集まりである。例えば、ブール型はtrueと**`false`の**値で構成される。整数も型を形成する。整数は**単純な型で**あり、その値にはサブパートが含まれないからである。銀行口座のレコードには通常、名前、住所、口座番号、口座残高などいくつかの情報が含まれる。このようなレコードは**集約**型や**複合型の**一例です。**データ**項目とは、ある型から値が引き出される情報またはレコードの一部である。データ項目は型の**メンバと**呼ばれる。

データ型とは、型と、その型を操作するための操作のコレクションの

ことである。例えば、整数変数は整数データ型のメンバーである。足し算は整数データ型の操作の一例である。

データ型の論理的な概念と、コンピュータ・プログラムにおける物理的な実装は区別する必要がある。例えば、リスト・データ型には、連結リストと配列ベースのリストという2つの伝統的な実装がある。したがって、リスト・データ型は連結リストまたは配列を使って実装することができる。配列」という用語も、データ型を指す場合と実装を指す場合があるという点で曖昧である。「配列」とは、コンピュータ・プログラミングでは一般的に、連続したメモリ・ロケーションのブロックを意味し、各メモリ・ロケーションには固定長のデータ項目が1つ格納される。この意味で、配列は物理的なデータ構造である。しかし、配列は、データ項目の（一般的に同種の）コレクションから構成される論理データ型を意味することもあり、各データ項目はインデックス番号で識別される。配列はさまざまな方法で実装することができる。例えば

また、セクション12.2では、比較的少数の非ゼロ値のみを格納する大きな2次元配列であるスパース行列を実装するために使用されるデータ構造について説明します。この実装は、連続したメモリ位置としての配列の物理的な表現とはかなり異なる。

抽象データ型（ADT）は、データ型をソフトウェア・コンポーネントとして実現したものである。ADTのインターフェースは、型とその型に対する操作のセットで定義される。各操作の動作は、その入力と出力によって決定される。ADTは、データ型が*どのように*実装されるかを指定しない。これらの実装の詳細は、ADTのユーザーから隠され、外部からのアクセスから保護**さ**れる。

データ構造はADTの実装である。Javaのようなオブジェクト指向言語では、ADTとその実装が一緒になって**クラスを**構成する。ADTに関連する各操作は、**メンバ関数**または**メソッドによって**実装される。データ・アイテムが必要とする空間を定義する変数は、**データ・メンバーと**呼ばれる。オブジェクトはクラスのインスタンスであり、コンピュータ・プログラムの実行中に生成され、ストレージを占有するものである。

データ構造」という用語は、コンピュータのメインメモリーに保存されているデータを指すことが多い。関連用語の**ファイル**構造は、ディスクドライブやCDなどの周辺記憶装置上のデータの構成を指すことが多い。

例1.3 整数の数学的概念と、整数を操作するオペレーションが、データ型を形成する。Javaの**`int`**変数型は、抽象的な整数を物理的に表現したものである。**`int`**変数型は、**`int`**変数に作用する演算とともにADTを形成する。幸いなことに、**`int`変数の**実装は、**`int`**変数が格納できる値の範囲に制限があるため、抽象整数に完全に忠実ではありません。これらの制限が受け入れがたいことが判明した場合、ADT "integer "の他の表現を考案し、関連する演算に新しい

実装を使用しなければならない。

例1.4 整数リストのADTは、以下の操作を指定する：

- リストの特定の位置に新しい整数を挿入する。
- リストが空の場合は`true`を返す。
- リストを再初期化する。
- 現在リストにある整数の数を返す。
- リストの特定の位置の整数を削除する。

この記述から、各オペレーションの入力と出力は明らかなはずだが、リストの実装は指定されていない。

あるADTを使用するあるアプリケーションは、そのADTの特定のメンバ関数を2つ目のアプリケーションよりも多く使用するかもしれないし、2つのアプリケーションは様々な操作に対する時間要件が異なるかもしれない。このようなアプリケーションの要求の違いが、あるADTが複数の実装でサポートされる理由である。

例1.5 大規模なディスク・ベースのデータベース・アプリケーションでよく使われる2つの実装は、ハッシ ング（セクション9.4）とB^+ -ツリー（セクション10.5）である。どちらも効率的なレコードの挿入と削除をサポートし、完全一致クエリをサポートする。しかし、完全一致クエリでは、ハッシュの方がB^+ -treeよりも効率的である。一方、B^+ -treeは範囲クエリを効率的に実行できるが、ハッシュは範囲クエリでは絶望的に効率が悪い。したがって、データベース・アプリケーションが検索を完全一致クエリに限定する場合、ハッシュが好まれます。一方、アプリケーションでレンジクエリのサポートが必要な場合は、B^+ -treeが好まれる。このようなパフォーマンス上の問題があるにもかかわらず、どちらの実装も同じ問題を解決している。

ADTのコンセプトは、非競合的なアプリケーションであっても、重要な問題に焦点を当てるのに役立つ。

例1.6 自動車を運転するときの主な動作は、ステアリング、アクセル、ブレーキである。ほぼすべての乗用車では、ステアリング・ホイールを回して操舵し、アクセル・ペダルを踏んで加速し、ブレーキ・ペダルを踏んで制動する。このクルマの設計は、"ステア"、"アクセル"、"ブレーキ "という操作を持つADTとみなすことができる。2台のクルマは、エンジンの種類や前輪駆動と後輪駆動の違いなど、これらの操作を根本的に異なる方法で行うかもしれ

ない。しかし、ADTはドライバーに特定のエンジンや駆動方式の詳細を理解させることなく、統一された操作方法を提示するため、ほとんどのドライバーはさまざまな車を操作することができる。これらの違いは意図的に隠されている。

ADTのコンセプトは、成功するコンピューター・サイエンティストが理解しなければならない重要な原則の一例である。コンピュータ・サイエンスの中心的なテーマは、複雑さとそれを扱う技術である。人間は、オブジェクトや概念の集合体にラベルを割り当て、そのラベルを集合体の代わりに操作することで、複雑さに対処する。認知心理学者はこのようなラベルを**メタファーと**呼ぶ。特定のラベルは、他の情報や他のラベルと関連しているかもしれない。この集合体にもラベルが与えられ、概念とラベルの階層が形成される。このラベルの階層構造によって、私たちは不必要な細部を無視して重要な問題に集中することができる。

例1.7 特定の種類の記憶装置上のデータを操作するハードウェアの集合に「ハード・ドライブ」というラベルを貼り、コンピュータ命令の実行を制御するハードウェアに「CPU」というラベルを貼る。これらのラベルやその他のラベルは、"コンピューター "というラベルの下にまとめられている。今日、最小の家庭用コンピュータでさえ、何百万ものコンポーネントを持っているため、コンピュータがどのように動作するかを理解するには、何らかの形で抽象化する必要がある。

ADTを実装し、操作する複雑なコンピュータ・プログラムを設計するプロセスをどのように進めるかを考えてみましょう。ADTはプログラムのある部分で特定のデータ構造によって実装される。ADTを使用するプログラム部分を設計している間、あなたはデータ構造の実装を気にすることなく、データ型に対する操作の観点から考えることができます。このように複雑なプログラムに対する思考を単純化する能力がなければ、そのプログラムを理解したり実装したりすることはできないだろう。

例1.8 ディスク上に保存される比較的単純なデータベースシステムの設計を考えてみよう。通常、このようなプログラムのディスク上のレコードは、直接ではなくバッファプール（セクション8.3参照）を介してアクセスされる。可変長のレコードは、メモリ・マネージャ（セクション12.3参照）を使用して、ディスク・ファイル内の適切な場所を探してレコードを配置する。複数のインデックス構造（第10章参照）は通常、様々な方法でレコードにアクセスするために使用される。このように、それぞれ独自の責任とアクセス権限を持つクラスが連鎖している。ユーザーからのデータベースクエリは、インデックス構造を検索することで実装される。このインデックスは、バッファプールへのリクエストによっ

てレコードへのアクセスを要求する。レコードが挿入または削除される場合、このようなリクエストはメモリ・マネージャを経由し、メモリ・マネージャはバッファ・プールとやりとりしてディスク・ファイルにアクセスする。このようなプログラムは、人間のプログラマーが一度にすべての詳細を頭に入れておくには、あまりにも複雑すぎる。このようなプログラムを設計し実装する唯一の方法は、抽象化とメタファーを適切に使用することである。オブジェクト指向プログラミングでは、このような抽象化はクラスを使って処理される。

データ型には**論理**形式と**物理**形式がある。ADTの観点からのデータ型の定義は、その論理形式である。データ構造としてのデータ型の実装は、その物理的形式です。図 1.1 に、データ型の論理形式と物理形式の関係を示します。ADTを実装するとき は、関連するデータ型の物理的な形式を扱います。プログラムの他の場所で ADT を使用する場合は、関連するデータ型の論理形式を扱います。本書のいくつかのセクションでは、ADTの物理的な実装に焦点を当てています。

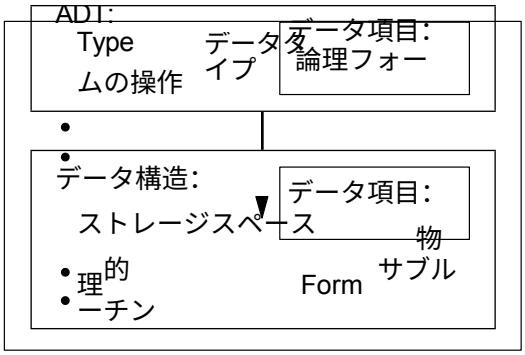

図1.1 データ項目、抽象データ型、データ構造の関係。ADT はデータ型の論理形式を定義する。データ構造はデータ型の物理的な形式を実装します。

与えられたデータ構造。他のセクションでは、上位タスクのコンテキストでデータ構造の論理ADTを使用する。

例1.9 あるJava環境で、リスト・クラスを含むライブラリが提供されるかもしれない。リストの論理形式は、クラスを定義するパブリック関数とその入力、出力によって定義される。リスト・クラスの実装について知っていることはこれだけかもしれませんし、これだけ知っていれば十分でしょう。リスト・クラスでは、さまざまな物理的実装が可能です。セクション4.1でいくつか説明します。

1.3 デザイン パターン

ADTよりも高い抽象度では、プログラムの設計、つまりオブジェクトとクラスの相互作用を記述するための抽象度がある。経験豊富なソフトウェア設計者は、ソフトウェア・コンポーネントを組み合わせるためのパターンを学び、再利用する。これらは**デザイン・パターンと**呼ばれるよう

になった。

デザイン・パターンは、繰り返し発生する問題に対する重要な設計概念を具体化し、一般化したものである。デザイン・パターンの第一の目的は、熟練した設計者が得た知識を新しいプログラマーに素早く伝達することである。もうひとつの目的は、プログラマー間の効率的なコミュニケーションを可能にすることである。トピックに関連する技術的な語彙を共有することで、設計上の問題について議論することがより容易になります。具体的なデザイン・パターンは、特定の設計上の問題が多くの文脈で繰り返し現れることを理解することから生まれます。デザイン・パターンは、実際の問題を解決するためのものです。デザイン・パターンはジェネリックスのようなものです。デザイン・パターンは、デザイン・ソリューションの構造を記述するものであり、どのような問題に対しても細部を補うことができる。デザインパターンはデータ構造に似ている：それぞれがコストとベネフィットを提供する。

トレードオフが可能であること。したがって、あるデザインパターンは、ある状況に内在する様々なトレードオフに適合するように、その適用にバリエーションを持つかもしれない。

このセクションの残りの部分では、この本の後半で使われる簡単なデザインパターンをいくつか紹介する。

1.3.1 フライ級

Flyweightデザインパターンは、次のような問題を解決するためのものだ。多くのオブジェクトを持つアプリケーションがある。これらのオブジェクトの中には、含まれる情報や果たす役割が同じものもあります。しかし、それらはさまざまな場所から到達しなければならず、概念的には本当に異なるオブジェクトです。同じ情報の重複が非常に多いため、そのスペースを共有することでメモリ・コストを削減する機会を利用したい。例えば、文書のレイアウトを表現する場合だ。文字 "C "は、その文字のストロークとバウンディングボックスを記述するオブジェクトで表現するのが合理的かもしれません。しかし、ドキュメント内の "C "が現れるすべての場所に、別々の "C "オブジェクトを作成することは避けたい。解決策は、"C "オブジェクト用の共有表現のコピーを1つ割り当てることです。そうすれば、与えられたフォント、サイズ、書体で "C "を必要とする文書内のすべての場所は、この単一のコピーを参照することになる。特定の書体の "C "を参照するさまざまなインスタンスをフライウェイトと呼ぶ。

ページ上のテキストのレイアウトをツリー構造で表現することができる。ツリーのルートはページ全体を表す。ページには複数の子ノードがあり、各列に1つずつあります。列ノードは各行の子ノードを持つ。そして、行には各文字の子ノードがあります。これらの文字の表現がフライウェイトである。フライウェイトには、共有シェイプ情報への参照が含まれ、そのインスタンスに固有の追加情報が含まれる場合もあります。

たとえば、"C "の各インスタンスには、ストロークとシェイプに関する共有情報への参照が含まれ、ページ上のその文字のインスタンスの正確な位置も含まれる場合があります。

フライウェイトは、セクション13.3で説明する、点オブジェクトのコレクションを格納するためのPR四分木データ構造の実装で使用される。PR四分木では、やはり葉ノードを持つ木がある。これらのリーフノードの多くは空の領域を表すため、格納される情報は空であるという事実のみである。これらの同一のノードは、フライウェイトの単一のインスタンスへの参照を使用して実装することで、メモリ効率を向上させることができます。

1.3.2 ビジター

ページ・レイアウトを記述するオブジェクトのツリーが与えられたとき、ツリー内のすべてのノードに対して何らかのアクティビティを実行したいと思うかもしれない。セクション5.2では、ツリー・トラバーサルについて説明する。ツリー・トラバーサルとは、ツリー内のすべてのノードを定義された順序で訪問する処理である。テキスト合成アプリケーションの簡単な例は、ツリー内のノード数を数えることです。

を生成する。また別の機会に、デバッグのためにすべてのノードのリストを表示したいと思うかもしれない。

私たちは、ツリー上で実行する傾向があるそのようなアクティビティごとに、個別のトラバーサル関数を書くことができる。より良い方法は、一般的なトラバーサル関数を書き、各ノードで実行されるアクティビティを渡すことである。このような構成がビジター設計パターンを構成する。ビジター設計パターンは、第5.2節（木の走査）と第11.3節（グラフの走査）で使用されます。

1.3.3 コンポジット

アクションのコレクションとオブジェクトタイプの階層の関係を扱うには、2つの基本的なアプローチがある。まず、典型的な手続き的アプローチを考えてみよう。ページレイアウト・エンティティの基本クラスがあり、特定のサブタイプ（ページ、列、行、図形、文字など）を定義するサブクラス階層があるとする。そして、そのようなオブジェクトの集合に対して実行されるアクションがあるとする（オブジェクトを画面にレンダリングするなど）。手続き設計のアプローチとしては、各アクションを、基本クラス型へのポインターをパラメーターとするメソッドとして実装する。各アクション・メソッドは、オブジェクトのコレクションを走査し、順番に各オブジェクトを訪れます。各アクション・メソッドには、コレクション内の各サブクラス（ページ、列、行、文字など）に対するアクションの詳細を定義するswitch文のようなものが含まれる。トラバーサルを一度だけ記述し、次にオブジェクトのコレクションに適用される可能性のある各アクションのためのビジターサブルーチンを記述する必要があるように、ビジターデザインパターンを使用することによって、コードをいくらか削減することができます。しかし、各ビジターサブルーチンには、可能性のある各サブクラスを処理するためのロジックが含まれていなければなりません。

ページ合成アプリケーションでは、ページ表現で実行したいアクティビティはほんのわずかしかない。オブジェクトを完全に詳細にレンダリングするかもしれない。あるいは、オブジェクトのバウンディングボックスだけを印刷する「下書き」レンダリングが必要かもしれません。オブジェクトのコレクションに適用する新しいアクティビティを思いついたら、既存のアクティビティを実装するコードを変更する必要はない。しかし、このアプリケーションでは、新しいアクティビティを追加することはあまりないだろう。対照的に、オブジェクト・タイプは数多く存在する可能性があり、私たちの実装に新しいオブジェクト・タイプを頻繁に追加するかもしれません。残念ながら、新しいオブジェクト・タイプを追加するには、各アクティビティを修正する必要があり、アクティビティを実装するサブルーチンでは、多くのサブクラスの動作を区別するために、かなり長いswitch文が必要になります。

別の設計としては、階層内の各オブジェクトのサブクラスに、実行される可能性のあるさまざまなアクティビティごとのアクションを具体化させる方法があります。各サブクラスは、各アクティビティ（フルレンダリングやバウンディングボックスレンダリングなど）を実行するコードを持つ。そして、コレクションにアクティビティを適用したい場合は、コレクション内の最初のオブジェクトを呼び出し、アクションを（そのオブジェクトのメソッド呼び出しとして）指定するだけです。ページレイアウトとその階層的なオブジェクトコレクションの場合、他のオブジェクトを含むオブジェクト（文字を含む行オブジェクトなど）は

は、それぞれの子に適したメソッドを呼び出します。この構成で新しいアクティビティを追加したい場合、すべてのサブクラスのコードを変更しなければならない。しかし、私たちのテキスト合成アプリケーションでは、このようなことは比較的まれです。対照的に、新しいオブジェクトをサブクラス階層に追加する（このアプリケーションでは、新しいレンダリング関数を追加するよりもはるかに可能性が高い）のは簡単だ。新しいサブクラスを追加しても、既存のサブクラスを変更する必要はありません。単に、新しいサブクラスで実行できる各アクティビティの動作を定義する必要があるだけです。

機能アクティビティをサブクラスの中に埋没させるこの2番目の設計手法は、Composite設計パターンと呼ばれる。Compositeデザイン・パターンの詳しい使用例はセクション5.3.1で説明します。

1.3.4 戦略

デザインパターンの最後の例では、より大きなアクティビティの一部として実行される可能性のある一連の代替アクションをカプセル化し、交換可能にします。再びテキスト合成の例を続けると、レンダリングしたい出力デバイスごとに、実際のレンダリングを行うための独自の関数が必要になる。つまり、オブジェクトはピクセルやストロークに分解されるが、ピクセルやストロークをレンダリングする実際の仕組みは出力デバイスに依存する。このレンダリング機能をオブジェクトのサブクラスに組み込みたくはない。代わりに、レンダリング動作を実行するサブルーチンに、その出力デバイスに適したレンダリングの詳細を実行するメソッドやクラスを渡したい。つまり、レンダリングタスクの詳細を達成するための適切な「ストラテジー」をオブジェクトに渡したいのだ。したがって、このアプローチはStrategyデザインパターンと呼ばれる。

Strategyデザインパターンは、一般化されたソート関数を作成するために使うことができます。ソート関数は追加のパラメータで呼び出すこ

とができます。このパラメータは、ソートされるレコードのキー値を抽出して比較する方法を理解するクラスです。この方法では、ソート関数はレコード型がどのように実装されているかの詳細を知る必要はありません。

デザイン・パターンを理解する上で最大の難関のひとつは、あるパターンと別のパターンが微妙にしか違わないことがあるということだ。例えば、コンポジット・パターンとビジター・パターンの違いについて混乱するかもしれない。その違いは、コンポジット・パターンは、トラバーサル・プロセスの制御をツリーのノードに与えるか、ツリー自体に与えるかという点にある。どちらのアプローチも、各ノードで実行されるアクティビティをカプセル化することで、トラバーサル関数を何度も書き直すことを避けるために、訪問者デザイン・パターンを利用することができる。

しかし、ストラテジー・デザイン・パターンも同じことをしているのではないだろうか? ビジター・パターンと戦略パターンの違いはもっと微妙だ。ここでの違いは、主に意図と焦点の違いです。ストラテジー・デザイン・パターンでもビジター・デザイン・パターンでも、アクティビティはパラメーターとして渡されます。strategyデザイン・パターンは、より大きなプロセスの一部であるアクティビティをカプセル化することに重点を置いています。

そのアクティビティを実行する異なる方法を代用できる。訪問者デザインパターンは、コレクションのすべてのメンバーにアクセスする汎用メソッド内でまったく異なるアクティビティを代用できるように、コレクションのすべてのメンバーで実行されるアクティビティをカプセル化することに重点を置いています。

1.4 問題、アルゴリズム、プログラム

プログラマーは通常、問題、アルゴリズム、コンピュータープログラムを扱う。これらは3つの異なる概念である。

問題：あなたの直感が示唆するように、問題とは実行すべきタスクである。インプットとアウトプットのマッチングという観点から考えるのが最適である。問題定義には、問題をどの*ように*解決するかについての制約を含めてはならない。解決方法は、問題が正確に定義され、十分に理解された後に初めて開発されるべきである。しかし、問題定義には、受け入れ可能な解決策によって消費される可能性のある資源に関する制約を含めるべきである。コンピュータが解くべき問題には、明示的であれ暗示的であれ、常にそのような制約がある。例えば、どのようなコンピュータ・プログラムも、利用可能なメイン・メモリとディスク・スペースしか使ってはならず、「妥当な」時間内に実行しなければならない。

問題は、数学的な意味での関数として捉えることができる。関数とは、入力（**領域**）と出力（**範囲**）のマッチングである。関数の入力は1つの値であったり、情報の集まりであったりする。入力を構成する値は関数の**パラメータと**呼ばれる。パラメータの値の特定の選択は、問題の**インスタンスと**呼ばれる。例えば、ソート関数の入力パラメータは整数の配列かもしれない。与えられたサイズと配列の各位置の特定の値を持つ整数の特定の配列が，ソート問題のインスタンスとなる．異なるインスタンスが同じ出力を生成するかもしれない。しかし，どのような問題イン

スタンスであっても，その特定の入力を使って関数を計算するときは常に同じ出力にならなければならない．

すべての問題は数学の関数のように振る舞うというこの概念は、コンピュータ・プログラムの振る舞いに対するあなたの直感と一致しないかもしれない。同じ入力値を2回に分けて与えると、2つの異なる出力が得られるプログラムを知っているかもしれない。例えば、一般的なUNIXのコマンド・ライン・プロンプトに「`date`」と入力すると、現在の日付が表示される。当然、同じコマンドを打ったとしても、日によって日付は異なる。しかし、日付プログラムの入力には、プログラムを実行するために入力するコマンド以上のものがあることは明らかである。日付プログラムは関数を計算する。言い換えれば、ある特定の日に、完全に指定された入力に対して正しく実行された日付プログラムから返される答えはひとつだけである。すべてのコンピュータ・プログラムにおいて、出力はプログラムの全入力セットによって完全に決定される。乱数発生器」でさえも、その入力によって完全に決定される（ただし、乱数発生システムの中には、乱数を受け入れることによってこれを回避しているものもあるようだ）。

ユーザーの制御を超えた物理的プロセスからの入力）。プログラムと関数の関係については、セクション17.3で詳しく説明する。

アルゴリズム： アルゴリズムとは、ある問題を解決するために実行される方法やプロセスのことである。問題を関数と見なせば、アルゴリズムとは、入力を対応する出力に変換する関数の実装である。一つの問題は、多くの異なるアルゴリズムによって解くことができる。あるアルゴリズムが解く問題は1つだけである（つまり、特定の関数を計算する）。本書では多くの問題を扱うが、そのうちのいくつかの問題については、複数のアルゴリズムを紹介する。重要な問題であるソートについては、10個近くのアルゴリズムを紹介する!

ある問題に対する複数の解法を知っていることの利点は、問題の特定のバリエーションや特定のクラスの入力に対しては、解法*A*が解法*B**よりも*効率的かもしれない一方で、別のバリエーションやクラスの入力に対しては、解法Bが*Aよりも*効率的かもしれないということである。例えば、ある並べ替えアルゴリズムは、整数の小さなコレクションを並べ替えるのに最適かもしれない（これは何度も並べ替える必要がある場合に重要である）。もう1つは、大きな整数のコレクションをソートするのに最適なアルゴリズムかもしれない。もう1つは、可変長の文字列のコレクションをソートするのに最適かもしれない。

定義によれば、あるものがアルゴリズムと呼ばれるのは、それが以下の性質をすべて備えている場合に限られる。

1. それは*正しくなければ*ならない。言い換えれば、各入力を正しい出力に変換し、望ましい関数を計算しなければならない。すべてのアルゴリズムが何らかの関数を実装していることに注意してください。なぜなら、すべてのアルゴリズムは、すべての入力を何らかの出力にマッピングするからです（その出力がプログラムのクラッシュであっても）。ここで問題となるのは、与えられたアルゴリズムが*意図された*関数を実装しているかどうかである。

2. それは一連の*具体的なステップ*で構成されている。具体的とは、そのステップで説明される動作が、アルゴリズムを実行しなければならない人間や機械によって完全に理解され、実行可能であることを意味する。また、各ステップは有限の時間で実行可能でなければならない。このように、アルゴリズムは、一連のステップを実行することで問題を解決するための「レシピ」を与えてくれる。ステップを実行する能力は、レシピを実行する対象が誰なのか、何なのかに依存します。例えば、料理本に載っているクッキーのレシピのステップは、人間の料理人に指示するには十分具体的であるが、自動クッキー製造工場のプログラミングには適していないと考えられるかもしれない。
3. 次にどのステップが実行されるのか、*曖昧であっては*ならない。多くの場合、それはアルゴリズム記述の次のステップである。選択（例えばJavaの`if`文）は通常、アルゴリズムを記述する言語の一部である。選択では、次にどのステップを実行するかを選択することができるが、選択を行う時点では選択プロセスは曖昧ではない。
4. *有限*個のステップで構成されていなければならない。もしアルゴリズムの記述が無限のステップ数で構成されていたら、私たちは決して望めないだろう。

それを書き留めることも、コンピュータ・プログラムとして実装することもできない。アルゴリズムを記述するためのほとんどの言語（英語や "擬似コード "を含む）は、反復として知られる繰り返し動作を実行するための何らかの方法を提供している。プログラミング言語における反復の例としては、Javaの**`while`**ループや**`for`**ループがある。反復は、実際に実行されるステップ数を入力によって制御しながら、短い記述を可能にする。

5. *終了*しなければならない。言い換えれば、無限ループに入ってはならない。

プログラム：私たちはしばしば、**コンピュータ・**プログラムとは、あるプログラミング言語におけるアルゴリズムのインスタンス、つまり具体的な表現であると考える。本書では、ほとんどすべてのアルゴリズムをプログラム、あるいはプロ グラムの一部という形で説明する。当然ながら、同じアルゴリズムのインスタンスであるプログラムは数多く存在する。なぜなら、現代のコンピュータ・プログラミング言語であれば、同じアルゴリズムのコレクションを実装するのに使うことができるからである（ただし、プログラミング言語によっては、プログラマーにとって使いやすいものもある）。説明を簡単にするために、「アルゴリズム」と「プログラム」という言葉をしばしば同じように使うが、実際には別々の概念である。定義によれば、アルゴリズムは、必要なときにプログラムに変換できるような十分な詳細を提供しなければならない。

アルゴリズムが終了しなければならないという要件は、すべてのコンピューター・プログラムがアルゴリズムの技術的定義を満たすわけではないということを意味する。オペレーティング・システムもそのようなプログラムの一つである。しかし、オペレーティング・システムの様々なタスク（それぞれに関連する入力と出力）を個々の問題として考えることができ、それぞれがオペレーティング・システム・プログラムの一部によって実装された特定のアルゴリズムによって解決され、その出力

が生成されるとそれぞれが終了する。

要約すると問題とは関数であり、入力から出力へのマッピングである。アルゴリズムとは、問題を解くためのレシピであり、そのステップは具体的で曖昧でない。アルゴリズムは正しく、有限の長さで、すべての入力に対して終了しなければならない。プログラムとは、プログラミング言語におけるアルゴリズムのインスタンス化である。

1.5 続きを読む

データ構造とアルゴリズムに関する初期の権威ある著作は、Donald E. Knuth著の「*The Art of Computer Programming*」シリーズで、第1巻と第3巻がデータ構造の研究に最も関連している [Knu97, Knu98]。データ構造とアルゴリズムに対する現代的な百科事典的アプローチは、Robert Sedgewickの*Algorithms* [Sed11]です。アルゴリズム、その設計、およびその解析に関する、非常に読みやすく優れた（しかし、より高度な）教育用入門書としては、*Introduction to Al- gorithms* を参照してください：Udi Manber [Man89]の「*Introduction to Al- gorithms: A Creative Approach*」を参照されたい。先進的で、循環論的なアプローチについては、Cormen, Leiserson, and Rivest著の*Introduction to Algorithms* [CLRS09]を参照してください。また、Steven S. Skienaの*The Algorithm Design Manual* [Ski10] は、*アルゴリズム設計のための*入門書である。

は、ウェブ上で利用可能なデータ構造やアルゴリズムの多くの実装へのポインタを提供する。

現代のプログラミング言語はすべて同じアルゴリズムを実装できる（より正確には、あるプログラミング言語で計算可能な関数は、ある標準的な機能を持つプログラミング言語であれば、どの言語でも計算可能である）という主張は、計算可能性理論の重要な結果である。この分野の簡単な入門書としては、James L. Hein, *Discrete Structures, Logic, and Computability* [Hei09]がある。計算機科学の多くは問題解決に専念している。実際、これが多くの人々をこの分野に引きつけるのである。George Polya著「*How to Solve It*」[Pol57]は、問題解決能力を向上させる方法に関する古典的な著作として知られている。より良い学生になりたいなら（一般的な問題解決能力も）、以下を参照されたい。フォルガーとルブランによる「*創造的問題解決のための戦略*」[FL95]、「*効果的な* Marvin Levine [Lev94]の「*Problem Solving*」、Arthur Whimbey and Jack Lochhead [WL99]の「*Problem Solving & Comprehen-sion*」、Zbigniew and Matthew Michaelewicz [MM08]の「*Puzzle-Based Learning*」などがある。

人間が複雑さを処理するためにメタファーの概念をどのように使うかについての良い議論については、ジュリアン・ジェインズ著『*二重心の崩壊における意識の起源*』[Jay90]を参照のこと。コンピュータサイエンスの教育やプログラミングに直接関係するものとしては、"Cogito, Ergo Sum！ Cognitive Processes of Students Dealing with Data Structures」（Dan Aharoni [Aha00]著）を参照。プログラミング・コンテクスト思考から、より高度な（そしてよりデザイン指向の）プログラミングに依存しない思考への移行に関する議論である。

より現実的なレベルでは、多くの人はより良いプログラムを書くためにデータ構造を学ぶ。プログラムが正しく効率的に動作することを期待するのであれば、まず自分自身や同僚が理解できるものでなければならない。KernighanとPikeの*The Practice of Programming* [KP99]は、良いコー

ディングやドキュメンテーションのスタイルなど、プログラミングに関する実践的な問題を数多く論じています。大規模なプログラムを書くことの難しさについての優れた（そして面白い！）入門書としては、古典的な*The Mythical Man-Month*を読んでください：Frederick P. Brooks [Bro95]による*ソフトウェアエンジニアリングに関するエッセイ*です。

Javaプログラマーとして成功したいのであれば、手近に良いリファレンスが必要である。David Flanaganの*Java in a Nutshell* [Fla05]は、Javaの基本に慣れている人にとって良いリファレンスとなる。

プログラム記述のメカニズムに熟達したら、次のステップはプログラム設計に熟達することだ。優れた設計を習得するのはどの分野でも難しいが、オブジェクト指向ソフトウェアの優れた設計は最も難しい芸術のひとつである。初心者の設計者は、よく知られ、よく使われているデザイン・パターンを学ぶことで、学習プロセスをジャンプ・スタートさせることができる。デザイン・パターンに関する古典的な参考書は『*デザイン・パターン*』である：*Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, and Vlissides著) [GHJV95]です（これは一般に「4人組のギャング」と呼ばれています）。残念ながら、この本を理解するのは非常に難しい、

というのも、その概念は本質的に難しいからである。デザイン・パターンについて論じたWebサイトや、「*デザイン・パターン*」の学習ガイドを提供しているWebサイトが数多くある。オブジェクト指向ソフトウェア設計について論じた他の2冊の本には、Dennis Ka- fura [Kaf98]の *Object-Oriented Software Design and Construction with* C++ とArthur J. Riel [Rie96]の*Object-Oriented Design Heuristics*がある。

1.6 エクササイズ

この章の練習問題は、本書の他の章とは異なる。これらの練習問題のほとんどは、次の章で解答されます。しかし、本書の他の部分で答えを調べては*いけません*。これらの練習問題は、後で取り上げる問題のいくつかを考えさせるためのものです。現在の知識でできる限り答えてください。

1.1 あなたが使ったことのあるプログラムの中で、許容できないほど遅いものを考えてみてください。そのプログラムを遅くしている特定の操作を特定しなさい。そのプログラムが十分に速く実行する他の基本的な操作を特定しなさい。

1.2 ほとんどのプログラミング言語には、組み込みの整数データ型がある。通常この表現には固定サイズがあり、整数変数に格納できる値の大きさに制限がある。コンピュータの利用可能なメイン・メモリの制限以外に）サイズの制限がなく、したがって整数を格納できる大きさに実際的な制限がない、整数の表現を説明しなさい。その表現を使って加算、乗算、指数の演算をどのように実行できるかを簡単に示せ。

1.3 文字列のADTを定義する。ADTは、文字列に対して実行可能な典型的な関数で構成され、各関数は入力と出力で定義される。次に、文

字列の2つの異なる物理表現を定義する。

1.4 整数リストのADTを定義する。まず、ADTが提供すべき機能を決める。例1.4がヒントになるでしょう。次に、抽象クラス宣言の形式で、関数、パラメータ、戻り値の型を示しながら、JavaでADTを仕様化します。

1.5 整数変数がコンピュータ上でどのように表現されるかを簡単に説明しなさい。(これらのことに馴染みがなければ、コンピュータサイエンス入門の教科書で1の補数と2の補数について調べてください)。なぜこのような整数表現がセクション1.2で定義されたデータ構造として適格なのでしょうか?

1.6 整数の2次元配列のADTを定義する。このような配列に対して実行できる基本的な操作を正確に指定する。次に、1000行1000列の配列を格納するアプリケーションを想像してください。

配列の値の10,000以上が非ゼロである。このような配列について、100万個の位置を必要とする標準的な2次元配列の実装よりもスペース効率のよい、2つの異なる実装を説明しなさい。

1.7 ソートプログラムの実装を任されたとしよう。目標はこのプログラムを汎用的なものにすることであり、どのようなレコードやキーの型が使われるかをあらかじめ定義したくないということである。この汎用化をサポートするために、単純なソート・アルゴリズム（挿入ソートや、あなたがよく知っている他のソートなど）を一般化する方法を説明しなさい。

1.8 配列に対する単純な逐次検索を実装することになったとしよう。問題は、検索をできるだけ一般的なものにしたいということだ。つまり、任意のレコードとキーの型をサポートする必要があります。この目標をサポートするために、検索関数を一般化する方法を説明しなさい。その関数が同じプログラムの中で、異なるレコード・タイプに対して複数回使用される可能性を考慮しなさい。同じレコードの異なるキー（同じタイプまたは異なるタイプの可能性がある）で関数を使用する必要がある可能性を考慮してください。例えば、学生データレコードを郵便番号、名前、給与、GPAで検索する場合があります。

1.9 すべての問題にアルゴリズムがあるのか?

1.10 すべてのアルゴリズムにJavaプログラムがあるのか?

1.11 家庭用コンピューターで動作するスペルチェック・プログラムの設計を考えてみよう。このスペルチェッカーは、20ページ以下の文書をすばやく処理できなければならない。このスペルチェッカーには約20,000語の辞書が付属していると仮定する。この辞書に対して、どのような原始的な操作を実行しなければならないか。

1.12 例題1.2で説明したような、アメリカの市や町の情報を含むデータベース・サービスの設計を任されたとする。このデータベースの実装

として可能なものを2つ提案しなさい。

1.13 各レコードに含まれるキーフィールドに関してソートされたレコードの配列が与えられたとする。指定されたキー値を持つレコードを見つけるために配列を検索する2つの異なるアルゴリズムを与えなさい。どちらが「良い」と考えるか、またその理由は何か。

1.14 整数の配列をソートするために提案された2つのアルゴリズムを比較するにはどうしたらいいでしょうか? 特に

(a) 2つのソートアルゴリズムを比較する基準として、コストの適切な尺度は何でしょうか?

(b) 2つのアルゴリズムが、このようなコスト対策のもとでどのようなパフォーマンスを発揮するかを判断するために、どのようなテストや分析を行いますか?

1.15 コンパイラやテキストエディタにとってよくある問題は、文字列中の括弧（またはその他の括弧）がバランスよく適切に入れ子になっているかどうかを判断することである。

例えば、文字列「(())()()」は適切にネストされた括弧のペアを含むが、文字列「)()()」は含まない。

(a) 文字列が適切に入れ子になったバランスの取れた括弧を含む場合に**真を**、そうでない場合に**偽を**返すアルゴリズムを与えよ。*ヒント*：合法的な文字列を左から右へスキャンしている間に、左括弧より右括弧の方が多くなることはない。

(b) 文字列が適切に入れ子になっておらず、バランスが取れていない場合、最初に現れた括弧の文字列中の位置を返すアルゴリズムを与えよ。つまり、余分な右括弧が見つかった場合はその位置を返し、左括弧が多すぎる場合は、最初に抜けた左括弧の位置を返す。文字列が適切にバランスよく入れ子になっていれば1を返す。

1.16 グラフは頂点の集合と辺の集合で構成され、各辺は2つの頂点を結ぶ。任意の頂点の組は、1つの辺によってのみ接続することができる。グラフの頂点と辺で定義される接続を表現する少なくとも2つの異なる方法を説明しなさい。

1.17 あなたが大企業の出荷事務員だとしよう。あなたは今、約1000枚の請求書を手渡されたところだ。請求書はそれぞれ右上に大きな数字が書かれた1枚の紙である。送り状は、この番号の小さいものから大きいものへと並べ替えなければならない。請求書を分類するためのさまざまな方法を、思いつく限り書き出しなさい。

1.18 約1000個の整数からなる配列を、値の小さいものから大きいものへ並べ替えるにはどうすればよいか。配列をソートするためのアプローチを少なくとも5つ書きなさい。Javaや擬似コードでアルゴリズムを書かないでください。各アプローチについて、それがどのように機能するかを1文か2文で書きなさい。

1.19 ソートされていない）配列中の最大値を求めるアルゴリズムを考えよう。次に、配列の中で2番目に大きな値を求めるアルゴリズムを

考えよう。どちらが実装が難しいだろうか? どちらが実行に時間がかかるか（実行される比較の数で測定）。次に、3番目に大きな値を求めるアルゴリズムを考えよう。最後に、中間の値を求めるアルゴリズムを考えなさい。これらの問題のうち、解くのが最も難しいのはどれでしょうか?

1.20 ソートされていないリストでは、リストの最後に新しい要素を追加することで、コンスタント・タイムの挿入が可能になる。残念なことに、キー値Xを持つ要素を検索するには、Xが見つかるまでソートされていないリストを逐次検索する必要があり、平均してリストの半分の要素を見る必要がある。一方、n個の要素を持つソート済み配列ベースのリストは、バイナリサーチを用いれば$\log n$の時間で検索できる。残念ながら、新しい要素を挿入するには多くの時間が必要である。なぜなら、ソートされた状態を維持したい場合、多くの要素が配列内でずれてしまう可能性があるからだ。対数nの時間で挿入と検索の両方をサポートするには、データをどのように整理すればよいだろうか?

2

数学的予備知識

この章では、本書を通して使用される数学的表記法、背景、テクニックを紹介する。この資料は主に復習と参考のために提供されている。後の章で馴染みのない表記法や数学的技法に出会ったときに、関連するセクションに戻るとよいだろう。

推定に関するセクション2.7は、多くの読者にとってなじみが薄いかもしれない。見積もりは数学的な技術ではなく、むしろ一般的な工学的技術である。なぜなら、推定されたリソース要件が問題の再ソース制約から大きく外れている提案された解決策は、即座に破棄することができ、より有望な解決策の分析に時間を割くことができるからである。

2.1 集合と 関係

数学的な意味での集合の概念は、コンピュータ・サイエンスに広く応用されている。アルゴリズムの記述や実装を行う際に、集合論の記法や技法が一般的に使用される。集合に関連する抽象化は、アルゴリズム設計の明確化や簡略化に役立つことが多いからだ。

集合は、区別可能な**メンバー**または**要素の**集まりである。メンバは通常、**基本型と**呼ばれる大きな集団から引き出される。集合の各メンバーは、基本型の**プリミティブ要素**であるか、集合そのものである。集合には重複の概念はない。基本型の各値は、集合に含まれるか、集合に含ま

れないかのどちらかである。例えば、**Pという**名前の集合は3つの整数7、**26**、42で構成される。この場合、**Pの**メンバ**は**、データ構造**であり**、基本型は整数である。

図2.1は、集合とその関係を表すのによく使われる記号を示している。この表記法の使用例をいくつか紹介しよう。まず2つの集合Pと**Qを**定義する。

$$\mathbf{P} = \{2, 3, 5\}, \qquad \mathbf{Q} = \{5, 10\}.$$

{1, 4}	1と4のメンバーで構成される集合 **集合元を**使った集合の定義
{x ∣ xは正の整数}である。	例：すべての正の整数の集合
x∈P	xは集合**P**
x ∉ **P**	**xは**集合**Pの**メンバではない
ø	ヌルまたは空集合 カーディナリティ：集合**Pの**サイズ
|*P*|	または集合**Pの**メンバー数
P⊆Q, **Q⊇P**	セットPはセット**Qに**含まれる、集合Pは集合**Qの**部分集合であり、集合Qは集合**Pの**上位集合である。
p∪q	セット・ユニオン **P**または**Qに**現れるすべての要素
p∩q p	交差点を設定する：**P** AND **Qに**現れるすべての要素
- q	セットの違い：集合**Pの**すべての要素が集合**Qに**含まれない

図2.1 セット表記。

$|\mathbf{P}| = 3$（Pには3つのメンバーがいるため）、$\mathbf{Q} = 2$（Qには2つのメンバーがいるため）。Pと**Qの**和（**P** ∪ **Q**と表記）は、**P**または**Qの**どちらかに含まれる要素の集合であり、$\{2, 3, 5, 10\}$である。**P** と **Q** の交差（**P** ∩ **Q** と表記）は，**P** と **Q** の両方に現れる要素の集合であり，これは $\{5\}$ である．**P** − **Q** と書かれた **P** と **Q** の差集合は、**P** には現れるが **Q** には現れない要素の集合で、$\{2, 3\}$.この例では、**Q** − **P** = 10 である。集合には順序の概念がないため，集合 $\{4, 3, 5\}$ は 集合 **P** と区別できない．同様に、集合4,3,4,5も**P**と区別できない。集合には重複要素の概念がないからだ。

集合Sの冪集合とは、集合**Sに対して**可能なすべての部分集合の集合である。**S** = {*a*, *b*, *c*}。**S**のべき集合は

$$\{\varphi, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

集合のように）順序がなく、重複した値を持つ要素の集まりを**バッグと**呼ぶ[1]バッグと集合を区別するために、バッグの要素を角括弧 [] で囲む。例えば、バッグ[3, 4, 5, 4]はバッグ[3, 4, 5]とは異なる、**は** セット{3, 4, 5, 4}と区別がつかない。しかし、袋[3, 4, 5、4] はバッグと区別がつかない[3, 4, 4, 5]。

[1]ここでバッグと呼ばれるオブジェクトは、**マルチリストと**呼ばれることもある。しかし、私は、サブリストを含む可能性のあるリストに対してマルチリストという言葉を使うことにしている（セクション12.1参照）。

シーケンスは順序を持つ要素の集まりであり、重複値を持つ要素を含むことがある。シーケンスはタプルや**ベクトルとも**呼ばれる。シーケンスには、0番目の要素、1番目の要素、2番目の要素.....と続く。シーケンスの要素を角括弧で囲んで示す。例えば、3, 4, 5, 4は数列である。3,5,4,4という数列は、次の数列とは異なることに注意。
配列 3, 4, 5, 4 は、配列 3, 4, 5 とは異なる。

集合**S**上の**関係**Rは、集合**Sから**順序付けられたペアの集合である。

$$\{\langle a, c\rangle, \langle b, c\rangle, \langle c, b\rangle\}。$$

は関係であり

$$\{\langle a, a\rangle, \langle a, c\rangle, \langle b, b\rangle, \langle b, c\rangle, \langle c, c\rangle\}。$$

は別の関係である。タプルx, yが関係Rにある場合、xRyのようなinfix表記を使うことができる。私たちは、自然数の小なり演算子(<)のような関係をよく使うが、これには1, 3や2, 23のような順序ペアは含まれるが、3, 2や2, 2は含まれない。このような関係を順序対で書くのではなく、通常、1 < 3と書くように、このような関係には減数記法を使う。

Rを集合**S**上の二項関係として、関係の性質を次のように定義する。

- すべてのa（**S**）に対してaRaならば、Rは**反射的で**ある。
- Rは、すべてのa,b∈**Sについて**、aRbのときはいつでも、bRaのときはいつでも**対称で**ある。
- Rは、すべてのa,b∈**Sに対して**、aRbとbRaのとき、$a=b$であれば、**反対称で**ある。
- Rは、すべてのa,b,c∈**Sに対して**、aRbとbRcのとき、aRcであれば**推移的で**ある。

例として、自然数の場合、<は反対称（aRbとbRaが存在しないため）で他動的、≤は反射的で反対称で他動的、=は反射的で対称（そして反対称！）で他動的である。人の場合、"is a sibling of "という関係は対称的かつ推移的である。ある人物を自分自身の兄弟であると定義す

れば、それは再帰的であり、ある人物を自分自身の兄弟でないと定義すれば、それは再帰的でない。

Rは、反射的、対称的、推移的であれば、集合**S**上の**同値関係**である。同値関係を使って、集合を**同値クラスに**分割することが≡できる。集合**Sの**分割とは、互いに不連続で和が**S**である部分集合の集まりである。集合上の同値類の表現方法についてはセクション6.2を参照。不連続集合の応用例はセクション11.5.2にあります。

例2.1 整数の場合、=は各要素を明確な部分集合に分割する同値関係である。言い換えれば、任意の整数*a*について、3つのことが成り立つ。

1. $a = a$、

2. $a = b$ならば、$b = a$である。

3. $a = b$かつ$b = c$ならば、$a = c$である。

もちろん、異なる整数a、b、cについて、a＝b、b＝a、b＝cとなるケースは存在しない。したがって、＝が対称的かつ推移的であるという主張は空虚な真である（これらの事象が起こる例は関係には存在しない）。しかし、対称性と推移性の要件に違反しないので、関係は対称的かつ推移的である。

例2.2 兄弟姉妹の定義を、ある人が自分自身の兄弟姉妹であるという意味に明確化すると、兄弟姉妹関係は人の集合を分割する同値関係である。

例2.3 修飾関数（次節で定義）を使って同値関係を定義できる。整数の集合に対して、モジュラス関数を用いて、2つの数xとyが、$x \bmod m = y \bmod m$の場合にのみ、その関係にあるような2項関係を定義する。このようにして使われるモジュラスは、整数上の同値関係を定義し、この再関係は整数をm個の同値類に分割するのに使えることがわかる。この関係が同値関係であるのは

1. すべてのxについて、$x \bmod m = x \bmod m$である；
2. もし$x \bmod m = y \bmod m$なら、$y \bmod m = x \bmod m$である。
3. $x \bmod m = y \bmod m$ かつ $y \bmod m = z \bmod m$ ならば、$x \bmod m = z \bmod m$.

二項関係は、反対称的かつ推移的であれば**部分順序と**呼ばれる[2]部分順序が定義された集合は**部分順序集合**または**posetと**呼ばれる。ある集合の要素xとyは、xRyかyRxのどちらかであれば、与えられた関係のもとで**比較可能**である。部分順序のすべての要素の対が比較可能であれば、その順序は**全順序**または**線形順序と**呼ばれる。$\leq$

例2.4 整数について、関係<と部分順序を定義する。同様に、≦は全順序であり、$x \neq y$となるような整数xとyの組に対して、x≦y かy≦xのどちらかである。

---

[2]すべての著者がこの定義を使っているわけではない。私は文献の中で少なくとも3つの大きく異なる定義を見てきた。私は、<と≤の両方が整数の部分位数を定義するものを選んだ。

例2.5 整数のべき集合では、部分集合演算子は部分順序を定義する（反対称的かつ推移的だから）。例えば、{1, 2} ⊆ {1, 2, 3}。しかし、集合 1, 2 と 1, 3 は部分集合演算子では比較できない。したがって，部分集合演算子は整数の冪集合の全順序を定義しない．

2.2 その他 表記

単位： 単位には以下の表記を使う。"B "はバイト、"b "はビット、"KB "はキロバイト（$2^{10} = 1024$バイト）、"MB "はメガバイト（2^{20} バイト）、"GB "はギガバイト（2^{30} バイト）、"ms "はミリ秒（ミリ秒は1秒の$\frac{1}{1000}$）を表す。2の累乗を表す場合、数値と単位略号の間にはスペースを入れない。したがって、25ギガバイトのディスク・ドライブ（1ギガバイトは2^{30}バイトとする）は、"25GB "と表記される。スペースは、10進数で表記する場合に使用します。2000ビットは "2Kb "と書き、"2Kb "は2048ビットを表す。2000ミリ秒は2000ミリ秒と書く。この本では、大容量のストレージはほとんど常に2の累乗で、時間は10の累乗で測られることに注意してください。

階乗関数： **階乗**関数は、nが0より大きい整数の場合、$n!$ したがって、$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. 特殊なケースとして、$0!$ 階乗関数は、n が大きくなるにつれて急激に大きくなる。階乗関数を直接計算すると
は時間のかかるプロセスである。
良い近似である。スターリングの近似式では、$n!$ $\approx \sqrt{2\pi n}(\frac{n}{e})^n$, ここで $e \approx 2.71828$ (e は自然対数系√の底である）。したがって
は n^n よりも遅く成長する（なぜなら $\sqrt{2\pi n}/e^n < 1$ であるため）、任意の正の整数定数 c に対して c^n よりも速く成長する。

順列： ある数列**S**の順列とは、**S**をある順序で並べたものである。例えば、1 から n までの整数の順列は、それらの値をある順序で並べたもので

ある。もし数列がn*個の*異なるメンバーを含むなら、数列の順列はn通りある。これは、順列の最初のメンバーにはn*個の*選択肢があり、最初のメンバーの各選択肢に対して、2番目のメンバーには$n-1$*個の*選択肢があるためである。つまり，各順列が同じ確率で選択されるように，n*個の*可能な順列のうちの1つを選択するのである．ランダムな順列を生成する簡単なJava関数は以下の通りである。ここで，シーケンスのn*個の*値は，0

[3]記号"≒"は "ほぼ等しい "を意味する。

関数**swap(A, i, j)**は配列**A**の要素iと**j**を交換し、**Random(n)**は0から*n* 1の範囲の整数値を返す（swapと**Random**の詳細については付録を参照）。

```
/**  配列Aの値をランダムに並べ替える* / static
<E> void permute(E[] A) {.
   for (int i = A.length; i > 0; i--) // 各iについて
swap(A, i-1, DSutil.random(i));       //A[i-1]とスワップする。
  }                                              // ランダムな要素
```

ブール変数：ブール変数とは、trueと**false**の2つの値のどちらかを取る変数（Javaでは**boolean**型）である。この2つの値は、それぞれ1と0に関連付けられることが多いが、そうでなければならない理由はない。0と**false**は論理的に異なる型のオブジェクトであるため、0とfalseの相関に依存するのはプログラミングのやり方として間違っている。

論理記法：記号論理やブール論理の表記法を使うことがある。*A* Bは "AはBを含意する "または "AならばB "を意味する。*A* ⇒ *B*は "A if and only if B "または "A is equivalent to B "を意味する。*A* ⇔ Bは "AまたはB "を意味する。∧（記号論理の文脈でも、ブール演算を実行するときでも有用）。*A* Bは "AとB "を意味する。Aと*A*はどちらも「Aではない」、またはAがブール変数である場合のAの否定を意味する。

階と天井：例えば、⌊3.4⌋は⌊3.0⌋と同じく3であり、⌊-3.4⌋は-4であり、⌊-3.0⌋は-3である。⌊0⌋=-3。 *x*の**天井**（⌈*x*⌉と書く）は実数値*x*をとり、最小の整数≧*x*を返す。例えば、⌈3.4⌉=4は⌊4.0⌋と同じだが、⌈-3.4⌉=⌊-3.0⌋=-3.

モジュラス演算子：モジュラス（または**mod**）関数は、整数の除算の余りを返す。余りの定義から、 *n* mod *m* は、*q* が整数で *r* < *m* のとき *n* = *qm* + *r* となる整数 *r* です。したがって、nとmが正の整数のとき、|*n* mod *m*|の結果は0から*m* 1の間でなければならない。例えば、5 mod 3 = 2; 25 mod 3 = 1, 5 mod 7 = 5, そして
5 mod 5 = 0.

qと*r*に値を割り当てる方法は、整数除算の解釈の仕方によって複数ある。最も一般的な数学的定義では、mod関数を$n \bmod m = n - m\ n/m$として計算する。この場合、3 mod 5 = 2となる。しかし、JavaやC++のコンパイラーは、通常、整数演算を計算するために、基礎となるプロセッサーのma-chine命令を使用する。つまり、$n \bmod m = n - m(\mathrm{trunc}(n/m))$となる。この定義では、$-3 \bmod 5 = -3$である。

残念なことに、多くのアプリケーションでは、これはユーザーが望むものでも期待するものでもない。例えば、多くのハッシュシステムでは、レコードのキー値に対して何らかの計算を行い、その結果をハッシュテーブルのサイズに乗じる。ここで期待されるのは、結果が負数ではなく、ハッシュテーブルへの正当なインデックスであることである。ハッシュ関数の実装者は、計算の結果が常に正であることを保証するか、あるいはその結果が負である場合にモジュロ関数の結果にハッシュテーブルサイズを追加しなければならない。

2.3 対数

値y*に対する*底b*の*対数とは、y*を*得るためにbをべき乗したものである。通常、これは$\log_b\ y = x$*と*書かれる。したがって、$\log_b\ y = x$ならば、$b^x = y$であり、$b\ b^{\log y} = y$である。

対数はプログラマーによく使われる。代表的な使い方を2つ紹介しよう。

例2.6 多くのプログラムは、オブジェクトの集合に対する符号化を必要とする。n *個の*異なる符号値を表現するのに必要な最小のビット数は? 答えは$\lceil \log_2\ n \rceil$ビットである。例えば、1000 個のコードを格納する場合、1000 個の異なるコード（10 ビットで 1024 個の異なるコード値）を格納するには、少なくとも $\log_2 1000 = 10$ ビットが必要です。

例2.7 値の小さいものから大きいものへとソートされた配列の中から、与えられた値を見つけるためのバイナリサーチ・アルゴリズムについて考えてみよう。バイナリサーチは、まず真ん中の要素を見て、探している値が配列の上半分にあるか下半分にあるかを判断する。その後、アルゴリズムは目的の値が見つかるまで、適切な部分配列を半分に分割し続ける。(バイナリサーチについてはセクション3.5で詳しく説明する。) サイズnの配列は、最後の部

分配列に1つの要素しか残らないまで、何回半分に分割できるだろうか? 答えは*[*$\log_2 n$|回*]*である。

本書では、ほとんどすべての対数の底は2である。これは、データ構造とアルゴリズムが物事を半分に分割したり、2進数ビットでコードを格納したりすることが多いためである。本書で$\log n$という表記を目にするときはいつも、$\log_2 n$を意味するか、あるいは漸近的に使われているので実際の底は問題ではない。2以外の底を使う対数は、底を明示する。

対数は、任意の正の値m、n、およびについて、以下の特性を有する。r、および任意の正の整数aとb。

1. $\log(nm) = \log n + \log m$。

2. $\log(n/m) = \log n - \log m$。

3. $\log(n^r) = r \log n$。

4. $\log_a n = \log_b n / \log_b a$.

最初の2つの性質は、2つの数の対数を足す（または引く）ことによって、2つの数の掛け算（または割り算）の対数を求めることができるというものです。[4]性質(3)は、単に性質(1)の拡張である。性質(4)は、変数nと任意の2つの整数定数*a*、*b*に対して、$\log_a n$と$\log_b n$は、*n*の値に関係なく、定数係数$\log_b a$だけ異なることを示す。本書のほとんどの実行時分析は、コストにおける定数因子を無視するタイプである。特性(4)は、このような分析では、対数の底を気にする必要がないことを示している。なお、$2^{\log n} = n$である。

対数を議論するとき、指数がしばしば混乱を招く。(3)の性質から、$\log n^2 = 2 \log n$ となる。（*n* の対数^2 ではなく）対数の2乗を示すにはどうすればよいだろうか？これは$(\log n)^2$ と書くこともできるが、$\log^2 n$ を使うのが伝統的である。一方、*n*の対数の対数を取りたい場合もあります。これは$\log \log n$と書く。

値≦*1*になるまでに、ある数の対数を何回とらなければならないかを知る必要がある場合は、特別な表記法が使われる。この量は$\log^* n$と表記される。例えば、$\log^* 1024 = 4$ というのは、$\log 1024 = 10$, $\log 10 \approx 3.33$, $\log 3.33 \approx 1.74$, $\log 1.74 < 1$ であり、合計4回の対数演算が必要だからである。

2.4 和算と再帰

ほとんどのプログラムにはループ構造が含まれている。ループを含むプログラムの実行時間コストを分析する場合、ループが実行されるたびにコストを合計する必要がある。これが**和算の**例である。和は、あるパラメータ値の範囲に適用される関数のコストの合計です。和は通常、次のような "シグマ "表記で書かれる：

$$\sum_{i=1}^{n} f(i) \text{ 。}$$

この表記は、$f(i)$の値をある範囲の（整数）値で合計していることを示す。式のパラメータとその初期値は、記号の下に示される。ここでは、iという記法で、パラメータが値 1 で始まることを示す。これはパラメー

タiの最大値を示している。したがって、この表記法は、iが1からnまでの整数の範囲にわたって$f(i)$の値を合計することを意味する。

---

[4]これらの性質が計算尺の背景にある。つの数の足し算は、2つの長さを足し合わせて、その長さの合計を測ることと見なすことができる。掛け算はそう簡単にはいかない。しかし、まず数をその対数の長さに変換すれば、それらの長さを足し合わせることができ、その結果の長さの逆対数が掛け算の答えとなる（これは単に対数の性質(1)である）。計算尺は、数値の対数の長さを測定し、これらの長さを表す棒をスライドさせて合計の長さを求め、最後にこの合計の長さを、結果の対数の逆数をとることによって正しい数値の答えに変換する。

$f(1)+f(2)+\cdots+f(n-1)+f(n)$と表記する。文中のシグマ表記は$\sum_{i=1}^{n} f(i)$とタイプセットされる。

ある和が与えられたとき，それを和と同じ値を持つ代数方程式に置き換えたいと思うことはよくある．これは**閉形式解として**知られており、和を閉形式解に置き換えるプロセスは次のように知られている。

を、和を**解くこととして**使うことができる。例えば、総和$\sum_{i=1}^{n} 1$は、単純に元圧力の"1"をn回合計したものである（iは1からnまでの範囲であることを覚えておいてほしい）。なぜなら

以下は、有用な和とその閉形式の解のリストである。

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{2.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6} = \frac{n(2n+1)(n+1)}{6}. \tag{2.2}$$

対数$\sum_{i=1}^{n}$

$$n = n \log n. \tag{2.3}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \text{for } 0 < a < 1. \tag{2.4}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \quad \text{for } a \neq 1. \tag{2.5}$$

式2.5の特殊ケースとして、

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}、 \tag{2.6}$$

そして

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \tag{2.7}$$

式2.7の補則として、

対数$\sum_{i=0}^{n}$

$$2^i = 2^{\log n+1} - 1 = 2n - 1. \tag{2.8}$$

最後に

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}. \tag{2.9}$$

1からnまでの逆数の和は**調和級数と**呼ばれ、次のように書かれる。

H_n、$\log_e n$と$\log_e n+1$の間の値を持つ。より正確には、nが大きくなるにつれて

に近づいている。

$$H_n \fallingdotseq \log_e n + \gamma + \frac{1}{2n}、 \tag{2.10}$$

ここでγはオイラー定数であり、その値は0.5772...

これらの等式のほとんどは数学的帰納法によって簡単に証明できる（セクション2.6.3参照）。残念ながら、帰納法は閉形式の解を導く助けにはならない。提案された閉形式解が正しいかどうかを確認するだけである。閉形式解を導くテクニックについてはセクション14.1で説明する。

再帰的アルゴリズムの実行時間は、再帰式で最も簡単に表現できる。なぜなら、再帰的アルゴ リズムの合計時間には、再帰呼び出しの実行時間が含まれるからである。**再帰**関係は、それ自身の1つ以上の（より小さな）インスタンスを含む式によって関数を定義する。典型的な例は階乗関数の再帰的定義である：

$$n! = (n - 1)! \text{ - } n > 1; \qquad 1! = 0! = 1.$$

再帰のもう一つの標準的な例は、フィボナッチ数列である：

$n > 2$の場合、$\mathrm{Fib}(n) = \mathrm{Fib}(n - 1) + \mathrm{Fib}(n - 2)$; $\mathrm{Fib}(1) = \mathrm{Fib}(2) = 1.$

この定義から、フィボナッチ数列の最初の7つの数は次のようになる。

1、1、2、3、5、8、13。

この定義には、Fib(*n*)の一般的な定義と、Fib(1)とFib(2)の基本ケースの2つの部分がある。同様に、階乗の定義には再帰的な部分と基本ケースがある。

再帰関係は再帰関数のコストをモデル化するためによく使われる。例えば、セクション2.5の関数**fact**がサイズ*n*の入力に対して必要とする乗算の回数は、$n = 0$または$n = 1$の場合（基本ケース）にはゼロとなり、$n = 1$ の値に対して**fact**を呼び出す場合のコストに1を加えたものとなる。これは以下の再帰を用いて定義できる：

$n > 1$の場合、$\mathbf{T}(n) = \mathbf{T}(n - 1) + 1$; $\mathbf{t}(0) = \mathbf{t}(1) = 0$.

総和と同様、我々は通常、漸化式を閉形式で置き換えることを望む。その1つの方法は、右辺に現れる**T**をその定義に置き換えて漸化式を**展開する**ことである。

例2.8 再帰式$\mathbf{T}(n) = \mathbf{T}(n - 1) + 1$を展開すると、次のようになる。

$$\begin{aligned}\mathbf{T}(n) &= \mathrm{T}(n - 1) + 1 \\ &= (\mathbf{T}(n - 2) + 1) + 1。\end{aligned}$$

漸化式は何段階でも展開できるが、目的は漸化式を総和で書き換えることができるようなパターンを検出することである。この例では

$$(\mathbf{T}(n-2)+1)+1=\mathbf{T}(n-2)+2$$

そして、漸化式を再び展開すると、次のようになる。

$$\mathbf{T}(n)=\mathbf{T}(n-2)+2=\mathbf{T}(n-3)+1+2=\mathbf{T}(n-3)+3$$

というパターンに一般化される。 $\bar{}\ i)+i$ *に一般化*する。

$$\begin{aligned}\mathbf{T}(n) &= \mathrm{T}(n-(n-1))+(n-1)\\ &= \mathrm{T}(1)+n-1\\ &= n-1.\end{aligned}$$

パターンを推測しただけで、これが正しい閉形式解であることを実際に証明したわけではないので、帰納的証明を使ってプロセスを完成させよう（例2.13参照）。

例2.9 少し複雑な再帰は次のとおりである。

$$\mathbf{T}(n)=\mathbf{T}(n-1)+n; \qquad T(1)=1.$$

この漸化式を数ステップ展開すると、次のようになる。

$$\begin{aligned}\mathbf{T}(n) &= \mathrm{T}(n-1)+n\\ &= \mathrm{T}(n-2)+(n-1)+n\\ &= \mathrm{T}(n-3)+(n-2)+(n-1)+n.\end{aligned}$$

そして、この再発には、次のようなパターンがあるようだ。

$$\begin{aligned}\mathbf{T}(n) &= \mathrm{T}(n-(n-1))+(n-(n-2))+\cdots+(n-1)+n\\ &= 1+2+\cdots+(n-1)+n.\end{aligned}$$

これは総和 $\sum_{i=1}^{n}$ iと等価である。

閉形式解。

漸近関係の閉形式解を求める技法については第14.2章で述べる。第14

章以前では、本書で漸化式が使用されることはあまりなく、対応する閉形式解とその導出方法の説明は、使用時に提供される。

2.5 再帰

アルゴリズムが**再帰的で**あるのは、それ自身を呼び出して作業の一部を行う場合である。このアプローチが成功するためには、"自分自身への呼び出し "は、元々試みられた問題よりも小さな問題でなければならない。一般に、再帰的アルゴリズムには2つの部分が必要である：再帰的呼び出しに頼ることなく解くことができる単純な入力を扱う**基本**ケースと、パラメータがある意味で元の呼び出しのものよりも基本ケースに「近い」アルゴリズムへの1つ以上の再帰的呼び出しを含む再帰的部分である。以下は、*n*の階乗を計算する再帰的Java関数である。*n*の値が小さい場合のfactの実行のトレースはセクション4.2.4に示されている。

```
/**  nを再帰的に計算して返す! */ static long
fact(int n) {
  // fact(20) は long に収まる最大の値 assert (n >= 0) && (n
  <= 20) : "n out of range";
  if (n <= 1) return 1; // 基本ケース：基本解を返す return n*
  fact(n-1);                // n > 1 の再帰的呼び出し
}
```

この関数の最初の2行が基本ケースとなる。*n*≦1の場合の基本ケースは問題の解を計算する。*n* > 1の場合、**fact**は*n* 1 の階乗を求める方法を知っている関数を呼び出す。もちろん、*n* 1の階乗を計算する方法を知っている関数は、たまたま**fact**自身である。しかし、アルゴリズムを書きながら、このことを難しく考える必要はない。再帰的アルゴリズムの設計は常にこのようにアプローチできる。まず基本ケースを書く。次に、1つまたは複数の、より小さい、しかし類似した部分問題の結果を組み合わせて問題を解くことを考える。もしあなたが書いたアルゴリズムが正しければ、より小さな部分問題を解くために（再帰的に）そのアルゴリズムに頼ることができる。成功の秘訣は再帰呼び出しが*どのように*部分問題を解くかを気にしないこと。再帰呼び出しが*どのように部分問題を解くかを気にしないこと*だ。再帰呼び出しが正しく解くこ

とを受け入れ、その結果を使って元の問題を正しく解くのだ。これ以上簡単なことがあるだろうか?

再帰は、日常の物理的な世界の問題解決には対応するものがない。新しい方法で問題を考える必要があるため、この概念を理解するのは難しいかもしれない。再帰を効果的に使うには、再帰呼び出し以降の再帰的プロセスを分析するのをやめるように訓練する必要がある。部分問題は自分で解決する。あなたは、基本的なケースと、部分問題をどのように組み直すかを心配するだけでよい。

階乗関数の再帰バージョンは、**`while`**ループを使うことで同じ効果が得られるため、不必要に複雑に思えるかもしれない。ハノイの塔 "と呼ばれる有名なパズルを題材にした再帰の例をもう一つ紹介しよう。この問題を解く自然なアルゴリズムには、複数の再帰呼び出しがある。**`while`**ループを使って簡単に書き換えることはできない。

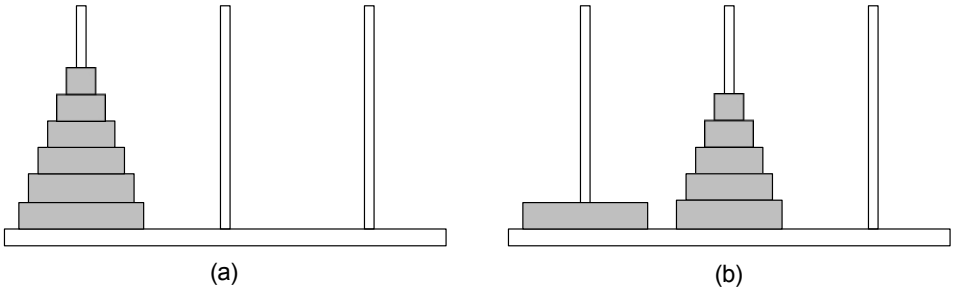

図2.2 ハノイの塔の例。(a) 6つのリングを持つ問題の初期条件。(b) 解を導くために必要な中間ステップ。

ハノイの塔パズルは、3本のポールと*n個の*リングから始まり、すべてのリングは一番左のポール（ポール1）からスタートします。図2.2(a)に示すように、リングの大きさはそれぞれ異なり、大きいリングから順に積み重ねられている。問題は，リングを左端のポールから右端のポール（ポール3というラベルが貼られている）まで，一連のステップで移動させることである．各ステップでは、あるポールの一番上のリングが別のポールに移動される。リングを動かす場所には制限がある：小さいリングの上にリングを移動させることはできない．

どうすればこの問題を解決できるのか? 難しく考えなければ簡単だ。その代わり、すべてのリングをポール1からポール3に移動させることを考える。まず一番下（一番大きい）のリングをポール3に移動させなければ、これを行うことはできない。そのためには、ポール3は空でなければならず、ポール1には一番下のリングしか置けない。残りの*n個のリング*は、図2.2(b)に示すように、ポール2の上に順番に積み上げなければならない。どうすればよいだろうか? 上の*n*個のリングをポール1からポール2に移動させる問題を解く関数Xが利用できると仮定する．次に，一番下のリングを極1から極3に移動する．最後に、残りの*n*個のリングをポール2からポール3に移動させるために、再び関数*X*を使用する。どちらの場合も、"関数*X*"は単純に、問題を縮小して呼び出されたハノイの塔関数である。

成功の秘訣は、Towers of Hanoiアルゴリズムに頼ることだ。Towers of Hanoiの部分問題が*どのように*解かれるのか、細かいことを気にする必要はない。つのことが実行されれば、それだけで解決する。第一に、再帰的な処理が永遠に続かないように、基本ケース（リングが1つしかない場合にどうするか）がなければならない。第二に、「ハノイの塔」の再帰的呼び出しは、より小さな問題、それも適切な形式（極の適切な名前変更を前提とした「ハノイの塔」問題の本来の定義を満たすもの）の問題を解くのにしか使えない。

再帰的ハノイの塔アルゴリズムの実装を以下に示す。関数**`move(start, goal)`は、スタート**地点から一番上のリングを取り出し、**ゴール**地点に移動させる。**もし`move`が**そのパラメーターの値を表示するとしたら、**`TOH`を**呼び出した結果は、問題を解決するリング移動命令のリストになる。

```
/**  ハノイの塔のパズルを解くために手を計算する。
     関数moveは、ある極から別の極へのディスクの実際の移動を行う（または
     表示する）。
     param n ディスクの数 @param
     start スタート極 @param goal ゴー
     ル極 @param temp もう一方の極 /.*
static void TOH(int n, ポールスタート, ポールゴール, ポールテンポ) { {.
  if (n == 0) return;              // 基本の場合
  TOH(n-1, start, temp, goal); // 再帰呼び出し: n-1move(start,
  goal);                                  // ボトムディスクをゴールに移動
  TOH(n-1,temp, goal, start); // 再帰呼び出し: n-1 リング
}
```

再帰をよく知らない人は、再帰が主にアルゴリズムの設計と記述を簡略化するためのツールとして使われていることを受け入れがたいかもしれない。なぜなら、再帰は関数呼び出しを伴い、**while**ループのような他の選択肢よりも一般的に高価だからである。しかし、再帰的なアプローチは、通常、第3章で議論された意味で合理的に効率的な アルゴリズムを提供する（しかし、必ずしもそうではない！ 練習問題2.11を参照）。 必要であれば、セクション4.2.4で説明するように、明確な再帰的解を後で修正して、より高速な実装を得ることができる。

多くのデータ構造は自然に再帰的であり、自己相似的な部分から構成されていると定義できる。木構造はその一例である。したがって、このようなデータ構造を操作するアルゴリズムは、しばしば再帰的に表現される。検索やソートのアルゴリズムの多くは、**分割と結合の**戦略に基づいている。つまり、問題をより小さな（類似した）部分問題に分割し、その部分問題を解き、その部分問題の解を組み合わせて元の問題の解を形成することによって解を見つける。このプロセスは多くの場合、再帰を使用して実装される。このように、再帰は本書を通して重要な役割を果たし、再帰関数の例も数多く示される。

2.6 数学的証明 テクニック

どのような問題解決にも、調査と議論という2つの明確な部分がある。学生たちは、教科書や講義の中で議論だけを見慣れすぎている。しかし、学校で（そして学校を卒業した後の人生で）成功するためには、その両方が得意である必要があり、この2つの段階のプロセスの違いを理解する必要がある。問題を解決するには、うまく調査しなければならない。つまり、問題に取り組み、解決策を見つけるまでやり抜くことだ。そして、その答えを依頼人（宿題や試験で答案を書くときの「依頼人」が指導者であれ、上司への報告書であれ）に伝えるためには、解決策を明確に伝えるような方法で論証する能力が必要である。

簡潔に論証の段階では、明確で論理的な論証を行う能力、つまり優れたテクニカルライティングのスキルが必要となる。

標準的な証明技法に精通していることは、このプロセスに役立ちます。良い証明の書き方を知っていると、多くの点で役立ちます。第一に、あなたの思考プロセスが明確になり、説明も明確になります。第二に、矛盾による証明や帰納法による証明など、標準的な証明の構造の一つを使えば、あなたも読者もその構造について共通の理解に基づいて作業することになります。そのため、読者はあなたの論証の構造をゼロから解読する必要がないため、読者があなたの証明を理解するための複雑さが軽減されます。

このセクションでは、一般的に使用される3つの証明技法を簡単に紹介する。(i) 直接証明、(ii) 矛盾による証明、(iii) 数学的帰納法による証明。

2.6.1 直接証明

一般に、**直接**証明は単なる "論理的説明 "である。直接証明は演繹による議論と呼ばれることもある。これは単に論理的な議論である。英語では "if ... then "のような言葉で書かれることが多いが、"$P \Rightarrow Q$ "のような論理記法で書かれることもある。記号的な論理表記を使わなくても、論理学の基本定理を利用して議論を構成することはできる。例えば、PとQが等価であることを証明したい場合、まず$P \Rightarrow Q$を証明し、次に$Q \Rightarrow$ Pを証明すればよい。

ある領域では、証明は本質的に、開始状態から終了状態への一連の状態変化である。形式的な述語論理はこのように見ることができ、様々な「論理のルール」を使って、1つの式から変更を加えたり、いくつかの式を組み合わせて目的地までのルートで新しい式を作ったりする。微積分学の入門クラスで積分の問題を解くためのシンボリックな操作は、高校の幾何学の証明と同様の精神を持っている。

2.6.2 矛盾による証明

定理や声明を*反証する*最も簡単な方法は、その定理の反例を見つけることである。残念ながら、定理が正しいことを証明するには、定理を支持する例の数だけでは不十分である。しかし、反例による反証とどことなく似たアプローチとして、「矛盾による証明」というものがある。矛盾によって定理を証明するには、まず定理が*偽*であると*仮定する*。次に、この仮定に由来する論理的矛盾を見つける。矛盾を見つけるために使われた論理が正しければ、矛盾を解決する唯一の方法は、定理が偽であるという仮定が間違っているに違いないと認識することである。つまり、定理は真でなければならないと結論づけるのである。

例2.10 ここに、矛盾による簡単な証明がある。

定理2.1 *最大の整数は存在しない。*

証明：矛盾による証明。

ステップ1.逆の仮定：最大の整数が*あると*仮定する。

それをB（"最大 "の意）と呼ぶ。

ステップ2。この仮定が矛盾を導くことを示せ：$C = B + 1$を考えよう。C*は*2つの整数の和なので整数である。また、$C > B$であり、Bは結局最大の整数ではない。したがって、矛盾に達した。我々の推論の唯一の欠陥は、定理が偽であるという最初の仮定である。よって、定理は正しいと結論づける。 *2*

関連する証明技法は、逆接の証明である。次のことが証明できる。$P \Rightarrow Q$*は*、(not Q)⇒(not P)の証明によって成立する。

2.6.3 数学的帰納法による証明

数学的帰納法は様々な定理の証明に使える。帰納法は、単純な部分問題から積み上げて問題を解くことを考えるように促すので、アルゴリズム設計を考える上でも役に立つ。帰納法は、再帰関数が正しい結果を生成することを証明するのに役立つ。再帰を理解することは、帰納法を理解するための大きな一歩となる。

アルゴリズム解析の文脈において、数学的帰納法の最も重要な用途の1つは、仮説を検証する方法である。第2.4節で説明したように、和や漸化式の閉形式の解を求める場合、まず特定の式が正しい解であることを推測したり、証拠を得たりすることがある。その式が本当に正しければ、その事実を帰納法で証明するのは簡単なことである。

数学的帰納法は、以下の2つの条件が真であれば、**Thrmは**パラメータn*の*任意の値（n c *の*2場合、cはある定数）に対して真であるとする：

1. **基本ケース：Thrmは**$n = c$*で*成立する。
2. **帰納法のステップ：**もし**Thrmが**$n - 1$について成り立つなら、**Thrmは**n*に*

ついて成り立つ。

基本ケースの証明は通常簡単で、定理中のnに1などの小さな値を代入し、定理を検証するために必要な簡単な代数学や論理学を適用する必要がある。帰納法の証明は簡単なこともあれば難しいこともある。帰納法の別の定式化は**強帰納法として**知られている。強帰納法の帰納ステップは次の通りである：

2a.帰納法のステップ： すべてのk、$c \leqq k < n$に対してThrmが成り立つなら、Thrmはnに対して成り立つ。

基本ケースの検証と併せて）帰納法のステップのいずれかの変形を証明することで、数学的帰納法による満足のいく証明が得られる。

帰納法の証明を構成する2つの条件は、**Thrmが**$n = 1$に対して成り立つという事実の延長として、**Thrmが**$n = 2$に対して成り立つことを証明するために組み合わされる。この事実は、条件(2)または(2a)と組み合わされ、**Thrmが**$n = 3$に対しても成り立つことを示す。このように、2つの条件が証明されれば、**Thrmは***nの*全ての値（基本ケースより大きい値）に対して成立する。

数学的帰納法がこれほど強力なのは（そしてほとんどの人にとって最初は謎めいているのは）、**Thrmが***nより*小さいすべての値に対して成り立つという*仮定を*、**Thrmが***nに対して*成り立つことを証明するための道具として利用できるからである。この仮定があることで、元の定理そのものに取り組むよりも、帰納法のステップの方が証明しやすくなる。帰納仮説に頼ることができることで、問題にもたらすことができる余分な情報が得られる。

再帰と帰納法には多くの共通点がある。どちらも1つ以上の基本的な事例に基づいている。再帰関数は、問題のより小さなインスタンスに対して答えを得るために、それ自身を呼び出す能力に依存する。同様に、帰納法の証明は、定理を証明するための帰納仮説の真理に依存している。帰納仮説は何もないところから出てくるものではない。定理自体が真である場合にのみ真であり、したがって証明の文脈の中では信頼できる。帰納仮説を使って仕事をすることは、再帰呼び出しを使って仕事をすることとまったく同じである。

例2.11 数学的帰納法による証明の例である。最初の*n個の*正の整数の和を$\mathbf{S}(n)$と呼ぶ。

定理2.2 $\boldsymbol{S}(n) = n(n + 1)/2.$

証明数学的帰納法によって証明する。

1. **基本ケースを確認する。** $n = 1$の場合、$\mathbf{S}(1) = 1(1 + 1)/2$であることを確認する。$1(1+1)/2=1$なので、式は基本ケースで正しい。
2. **帰納仮説を述べよ。**帰納仮説は

$$\mathbf{S}(n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)(n)}{2}.$$

3. 帰納仮説は、$\mathbf{S}(n-1) = (n-1)(n)/2$ であり、$\mathbf{S}(n) = \mathbf{S}(n-1) + n$であるから、$\mathbf{S}(n-1)$を代入して次が得られる。

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n-1} i + n = \frac{(n-1)(n)}{2} + n$$

$$= \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}.$$

したがって、数学的帰納法によって、

$$\mathbf{S}(n) = \sum_{i=1}^{n} i = n(n+1)/2.$$

2

この例で何が起こったかを注意深く見てみよう。$\mathbf{S}(n-1)$が登場すれば、帰納仮説を用いて$\mathbf{S}(n-1)$を$(n-1)(n)/2$に置き換えることができるからである。ここから、$\mathbf{S}(n-1) + n$が元の定理の右辺に等しいことを証明するのは簡単な代数学である。

例題2.12 誘導のための適切な変数の選択を説明する、帰納法によるもう一つの簡単な証明を示そう。最初の*n個の*正の奇数の和がn^2であることを帰納法で証明したい。まず、*n番目の*奇数を記述する方法が必要で、それは単に$2n-1$である。これによって定理を和とすることができる。

定理2.3 $\sum_{i=1}^{n} (2i - 1) = n^2$.

証明： $n = 1$の基本ケースでは、$1 = 1^2$が得られる。帰納仮説は

$$\sum_{i=1}^{n-1} (2i - 1) = (n - 1)^2.$$

最初の*n個の*奇数の和は、最初の$n-1$個の奇数の和に*n*番目の奇数を足したものである。以下の2行目では、帰納仮説を用いて部分和（1行目の括弧内）を閉形式の解に置き換える。後は代数学がやってくれる。

$$\begin{aligned}
\sum_{i=1}^{n} (2i - 1) &= \left(\sum_{i=1}^{n-1} (2i - 1)\right) + 2n - 1 \\
&= [(n-1)^2] + 2n - 1 \\
&= n^2 - 2n + 1 + 2n - 1 \\
&= n^2.
\end{aligned}$$

44 したがって、数学的帰納法により、$\sum_{i=1}^{n}(2i-1)=n^2$.

例題2.13 この例題は、帰納法を用いて、漸化式の閉形式解が正しいことを証明する方法を示す。

定理2.4 *再帰関係* $\mathbf{T}(n) = \mathbf{T}(n\ 1)+1;\ \mathrm{T}\ (1) = 0$ *は閉形式解* $\mathbf{T}(n) = n\ 1$ *を持つ。*

証明 基本ケースを証明するために、$\mathbf{T}(1) = 1\ 1 = 0$であることを観察する。帰納仮説は、$\mathbf{T}(n\ 1) = n\ 2$である。再帰の定義と帰納仮説を組み合わせると、すぐに次のことがわかる。

$$\mathbf{T}(n) = \mathbf{T}(n\ -\ \ 1) + 1 = n\ -\ \ 2 + 1 = n\ -\ \ 1$$

こうして、数学的帰納法によって定理が正しいことが証明された。 *2*

例2.14 この例では、和や他の方程式を用いずに帰納法を用いている。また、より柔軟な基本ケースの使い方も示している。

定理2.5 *2¢と5¢の切手は、どのような値でも形成することができる。* ≥ 4*).*

証明： この定理は、値1と値3では成立しないので、値4の問題を定義する。4を基本ケースとして、4¢の値は2¢の切手2枚から作ることができる。帰納仮説は、2¢と5¢の切手の組み合わせから$n\ 1$の値を作ることができるというものである。ここで帰納仮説を用いて、2¢と5¢の切手から値nを得る方法を示す。値$n\ 1$の構成に5¢切手が含まれているか、含まれていないかのどちらかである。もしそうなら、5¢切手を3枚の2¢切手に置き換える。そうでない場合、そのメイクアップには少なくとも2枚の2¢切手が含まれていなければならない（少なくともサイズ4で、2¢切手しか含まれていないため）。この場合、2枚の2¢切手を1枚の5¢切手に交換する。どちらの場合でも、2¢切手と5¢切手で構成されるnの

価値がある。よって、数学的帰納法により、定理は正しい。 *2*

例2.15 強帰納法を使った例である。

定理2.6 n > 1の*とき、nはある素数で割り切れる。*

証明： 2は素数2で割り切れる。帰納仮説は、*任意の*値a（$2 \leq a < n$）はある素数で割り切れるというものである。nが素数の場合、nはそれ自体で割り切れる。nが素数でない場合、$n = a \times b$で、aとbはともに以下の整数である。

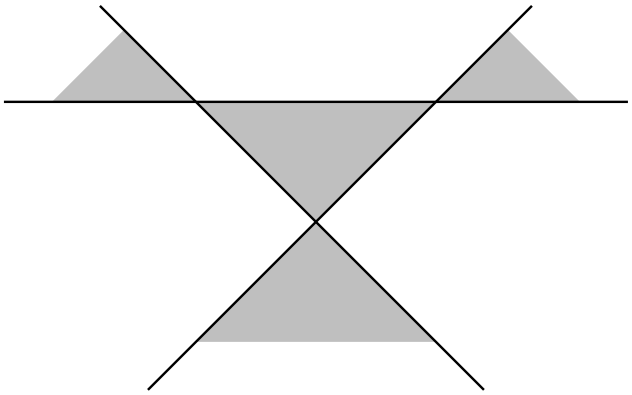

図2.3 平面上の3本の線によって形成される領域の2色塗り。

帰納仮説によれば、aはある素数で割り切れる。その素数はnも割り切らなければならない。したがって、数学的帰納法により、

定理は正しい。 *2*

数学的帰納法の次の例は、幾何学の定理を証明するものである。また、帰納法の証明の標準的な技法である、*n個の*物体を取り出して、帰納法の仮説を使うためにある物体を取り除くという技法も説明する。

例2.16 領域の集合の**2色使いを**、一辺を共有する2つの領域が同じ色を持たないように、 各領域に2色のうちの1色に署名する方法として定義する。例えば、チェス盤は2色である。図2.3は、3本の線を持つ平面の2色使いを示している。ここでは、黒と白の2色を使うと仮定する。

定理2.7 *平面上の*n本の*無限直線によって形成される領域の集合は2色になりうる。*

証明： 平面上に1本の無限直線がある場合を考える。この線は平面を2つの領域に分割する。片方の領域を黒く、もう片方を白く着色すれば、有効な2色になる。帰納仮説は、n 1本の無限直線によって形成される領域の集合は2色に着色できるというものである。*nについての*定理を証明するために、*n本の*線のどれかが取り除かれ

たときに残るn 1*本の*線によって形成される領域の集合を考える。帰納法によって、この領域の集合は2色になりうる。さて、n*番目の*線を戻す。これは平面を2つの半平面に分割し、それぞれの半平面は（独立に）、n 1本の線を持つ平面の2色塗りから継承された有効な2色塗りを持つ。残念なことに、n *番目の*線によって新しく分割された領域は、2 色のルールに違反する。n番目の線の片側にあるすべての領域を取り、それらの色付けを逆にする（そうしても、この半平面はまだ2色である）。n番目の線によって分割された領域は正しく2色になる。

この場合、線の片側の領域は黒に、もう片側の領域は白になる。したがって、数学的帰納法により、平面全体が2色になる。 *2*

定理2.7と定理2.5の証明を比べてみよう。定理2.5では、大きさn 1の切手の集合（帰納仮説により、これは望ましい性質を持たなければならない）を取り出し、そこから望ましい性質を持つ大きさnの集合を「作った」。したがって、我々は、望みの性質を持つサイズnの切手のコレクションの存在を証明した。

定理2.7では、*n本の*線の*任意の*集合が望ましい性質を持つことを証明しなければならない。したがって、我々の戦略は、*任意のn本の*線の集合を取り、それを「再デュース」して、帰納仮説に一致するため、望ましい性質を持つに違いない線の集合を作ることである。そこからは、元の漸化式を再変換しても目的の性質が保たれることを示せばよい。

これとは対照的に、サイズn 1の行の集合からサイズnの行の集合へと"構築 "しようとした場合、何が必要になるかを考えてみよう。任意のn行の集合からそれより小さいものへと縮小することで、この問題を回避することができる。

このセクションの最後の例では、再帰関数が正しい結果を生成することを証明するために帰納法をどのように使うかを示す。

例題2.17 関数 **`fact`** が確かに階乗関数を計算することを証明したい。このような証明には2つのステップがある。つは関数が必ず終了することを証明すること。もう1つは、関数が正しい値を返すことを証明することである。

定理2.8 *関数ファクトはどのようなnの値でも終了する。*

証明： 帰納仮説は、ファクトはn *1* で終了する。nには2つの可能性がある。一つは、nが$\leq$12である可能性である。この場合、**ファクトは**アサーション・テストに失敗するため、直接終了する。そうでなければ、factは**`fact(n-1)`**を再帰的に呼び出す。帰納仮

説により、**fact(n-1)は**終了しなければならない。 2

定理2.9 *関数factは、0から12の範囲内の任意の値に対して階乗関数を計算する。*

証明帰納仮説は、**fact(n-1)**は$(n-1)$の正しい値を返すということである。正当な範囲内の任意の値nに対して、**fact(n)は**n * **fact(n-1)**を返す。帰納仮説により、**fact(n-1)** $= (n-1)! = n!$であるから、**fact(n)が**正しい結果を返すことが証明された。 2

同じようなプロセスで、多くの再帰的プログラムの正しさを証明することができる。一般的な形は、基本ケースが正しく実行されることを示し、帰納仮説を用いて再帰ステップも正しい結果を生成することを示す。その前に、関数が常に終了することを証明しなければならない。

2.7 推定

コンピューター・サイエンスのトレーニングで身につけることができる最も役に立つライフ・スキルのひとつは、素早く見積もりを行う能力です。これは「back of the napkin」や「back of the envelope」と呼ばれることもあります。どちらのニックネームも、大まかな見積もりしか行わないことを示唆しています。見積もり技法は工学カリキュラムの標準的な部分ですが、コンピュータサイエンスでは軽視されがちです。見積もりは、問題の厳密で詳細な分析に取って代わるものではありませんが、厳密な分析が必要とされるときを示す役割を果たします：最初の見積もりで解決策が実行不可能であることがわかった場合、それ以上の分析はおそらく不要である。

見積もりは、以下の3つのステップで正式に行うことができる：

1. 問題に影響を与える主要なパラメータを決定する。
2. パラメータと問題を関連付ける方程式を導く。
3. パラメータの値を選択し、方程式を適用して推定解を得る。

見積もりをするとき、その見積もりが妥当であると自分を安心させる良い方法は、2つの異なる方法で見積もりをすることである。一般的に、あるシステムから何が出てくるかを知りたい場合、それを直接推定してみるか、あるいは（入ったものは必ず後で出てくると仮定して）システムに何が入るかを推定することができる。両方のアプローチが（それぞれ独立に）同じような答えを出すのであれば、それは推定値に対する信頼につながるはずである。

計算する際は、単位が合っていることを確認してください。例えば、フィートとポンドを足してはいけません。結果が正しい単位であることを確認してください。計算のアウトプットはインプットと同じだけ良いということを常に念頭に置いてください。ステップ3の入力パラメータの評価が不確かであればあるほど、出力値も不確かなものになります。しかし、概算計算は多くの場合、1桁以内、あるいはおそらく2倍以内の答えを得ることだけを目的としている。推定を行う前に、25%以内、2倍以内など、許容できる誤差の範囲を決めておく必要があります。推定値が誤差の範囲内に収まると確信したら、そのままにしておきましょう！目的に対して必要以上に正確な推定をしようとしないこと。

例2.18 100万ページの本を収納するのに必要な図書館の本棚の数は?

私は、500ページの本には

図書館の本棚の1インチ（どんな便利な本でも大きさを見るのに役立つだろう）、100万ページに対して約200フィートの本棚スペースがあることになる。棚の幅が4フィートだとすると、50枚の棚が必要だ。本棚に5つの棚があるとすると、図書館の本棚は約10個分となる。この結論に達するために、私は1インチあたりのページ数、棚の幅、本棚の棚の数を見積もった。どの推定値も正確ではないだろうが、私の答えは2分の1以内で正しいと確信している。(これを書いた後、バージニア工科大学の図書館に行き、本物の本棚を見た。それらは幅が3フィートほどしかなかったが、通常7つの棚があり、合計21棚フィートあった。つまり、本棚の容量については10%以内で正しかった。私が選択した数値のうち、ひとつは高すぎ、もうひとつは低すぎたため、誤差が相殺されたのだ)。

例2.19 ガソリンあたり20マイル走る車と、30マイル走るが3000ドル高い車とでは、どちらが経済的か。一般的な車の年間走行距離は約12,000マイルです。ガソリン代が1ガロンあたり3ドルだとすると、年間のガソリン代は効率の悪い車で1800ドル、効率の良い車で1200ドルとなる。3000ドルを銀行に投資した場合の回収額などの問題を無視すれば、価格差を埋 め る のに5年かかることになる。この時点で、買い手は価格だけが判断基準なのか、5年の投資回収期間が許容範囲なのかを決めなければならない。当然ながら、より多く車を運転する人の方がより早く差額を埋められるし、ガソリン価格の変動も結果に大きく影響する。

例2.20 スーパーマーケットで1週間分の買い物をするとき、レジで支払う金額 を概算できますか? 一つの簡単な方法は、各商品の値段を1ドル単位で四捨五入し、商品をカートに入れながら、この

値を頭の中で合計することです。そうすれば、本当の合計金額から数ドル以内の答えが出るでしょう。

2.8 さらに読む

この章で扱われるトピックのほとんどは、離散数学の一部と考えられている。この分野の入門書としては、Susanna S. Epp著 *Discrete Mathematics with Applications* [Epp10]がある。コンピュータ科学者に有用な多くの数学的トピックを高度に扱ったものとしては、*Concrete Mathematics*がある：*A Foundation for Computer Science (*Graham, Knuth, Patashnik著) [GKP94]がある。

本書で使用されているコンピュータの記憶装置の単位を示す基準については、*IEEE Spectrum*誌1995年2月号[Sel95]の "Technically Speaking "を参照のこと。

ウディ・マンバー著『*アルゴリズム入門*』[Man89]は、アルゴリズムを開発する技術として数学的帰納法を多用している。

再帰について詳しくは、Eric S. Roberts著「*Thinking Recursively*」[Rob86]を参照されたい。再帰を正しく学ぶためには、LISPやSchemeというプログラミング言語を学ぶ価値がある。特に、FriedmanとFelleisenの "Little "な本（*The Little LISPer*[FF89]や*The Little Schemer*[FFBS95]など）は、言語を教えるだけでなく、再帰的な考え方を教えるように作られている。これらの本は面白い読み物でもある。

数学的証明の書き方については、ダニエル・ソロウの*How to Read and Do Proofs* [Sol09]が良い。一般的な数学的問題解決の能力を向上させるには、Paul Zeitz著の*The Art and Craft of Problem Solving* [Zei07]を参照してください。Zeitzは、セクション2.6で紹介した3つの証明技法や、問題解決における調査と議論の役割についても論じています。

推定技法については、John Louis Bentley著の「*The Back of the Envelope*」と「*The Envelope is Back*」という2つのProgramming Pearls [Ben84, Ben00, Ben86, Ben88]を参照されたい。*Genius: The Life and Science of Richard Feynman* by James Gleick [Gle92]は、包絡線計算が原子爆弾の開発者や現代の理論物理学一般にとっていかに重要であったかを教えてくれる。

2.9 エクササイズ

2.1 以下の各関係について、その関係が反射的、対称的、反対称的、他動的の各特性を満たす、あるいは満たさない理由を説明しなさい。

(a) "isBrotherOf "である。

(b) "isFatherOf "である。

(c) 関係 $R = \{(x, y) \mid x^2 + y^2 = 1$ 実数xとyの場合。

(d) 関係 $R = \{(x, y) \mid x^2 = y^2$ 実数xとyについて。

(e) 関係 $R = \{(x, y) \mid x \bmod y = 0 \text{ for } x, y \in \{1, 2, 3, 4\}$.

(f) 整数の集合上の空関係 $\emptyset$ （すなわち、それが真となる順序ペアを持たない関係）。

(g) 空集合上の空関係 $\emptyset$ （すなわち、それが真となる順序ペアを持たない関係）。

2.2 次の各関係について、それが同値関係であることを証明するか、同値関係でないことを証明せよ。

(a) a と b の整数について、$a + b$ が偶数である場合に限り、$a \equiv b$ となる。

(b) a と b の整数について、$a + b$ が奇数である場合に限り、$a \equiv b$ となる。

(c) ゼロでない有理数aおよびbについて、$a \times b > 0$の場合に限り、$a \equiv b$となる。

(d) 非ゼロ有理数aおよびbについて、 $\equiv$ 、a/bが整数である場合に限る。

(e) 有理数aとbについて、a-bが整数である場合に限り、$a \equiv b$となる。

(f) 有理数aとbに対して、$a - b \leq 2$のときのみ、$a \equiv b$となる。

2.3 次の各関係が部分順序であるかどうかを述べ、その理由またはそうでない理由を説明しなさい。

(a) "isFatherOf "である。

(b) "isAncestorOf "を人々の集合に適用する。

(c) "isOlderThan "である。

(d) 人の集合に "isSisterOf"。

(e) 集合$\{a, b\}$上の$\{\langle a, b\rangle, \langle a, a\rangle, \langle b, a\rangle\}$。

(f) $\{1, 2, 3\}$の集合$\{\{2, 1\}, \}\}$上の$\{\hookrightarrow Ps_27E8, 1\rangle, \}$。

2.4 n個の要素を持つ集合に定義できる全順序はいくつあるか？答えを説明しなさい。

2.5 整数の集合のADTを定義する（集合には重複要素の概念がなく、順序の概念もないことに注意）。ADTは、集合のメンバシップを制御したり、サイズをチェックしたり、与えられた要素が集合内にあるかどうかをチェックしたりするために、集合に対して実行できる関数で構成されなければならない。各関数は、入力と出力で定義します。

2.6 整数のバッグのADTを定義しなさい（バッグは二個を含むことができ、順序の概念を持たないことに注意）。ADTは、メンバシップの制御、サイズのチェック、与えられた要素が集合に含まれるかどうかのチェックなど、バッグに対して実行可能な関数で構成されなければならない。各関数は入力と出力で定義する。

2.7 整数のシーケンスのADTを定義する（シーケンスは重複を含む可能性があり、要素の位置の概念をサポートしていることに注意）。ADTは、メンバシップの制御、サイズのチェック、指定された要素が集合内にあるかどうかのチェックなど、シーケンスに対して実行可能な関数で構成されなければならない。各関数は入力と出力で定義する必要があります。

2.8 ある投資家が3万ドルを株式ファンドに預けた。10年後、口座の価値は69,000ドルになった。対数と反対数を用いて、年平均増加率を計算する公式を示せ。次に、その計算式を使って、このファンドの年平均増加率を求めよ。

2.9 セクション2.5の階乗関数を再帰を使わずに書き直せ。

2.10 セクション2.2のランダム順列生成器の`for`ループを再帰関数として書き直す。

2.11 以下はフィボナッチ数列を計算する簡単な再帰関数である：

```
/**  再帰的にn番目のフィボナッチ数を生成して返す /*
静的長さ fibr(int n) { {.
  // fibr(91) は long に収まる最大の値 assert (n > 0) &&
  (n <= 91) : "n out of range";
  if ((n == 1) || (n == 2)) return;    // 基本ケース return
  fibr(n-1) + fibr(n-2);                //再帰呼び出し
}
```

このアルゴリズムは非常に遅く、**Fibr**を合計Fib(*n*)回呼び出すことになる。これは次の反復アルゴリズムと対照的である:

```
/**  反復的にn番目のフィボナッチ数を生成して返す /*
静的長さ fibi(int n) { {.
  // fibr(91) は long に収まる最大の値 assert (n > 0) &&
  (n <= 91) : "n out of range";
  long curr, prev, past;
  if ((n == 1) || (n == 2)) return 1;
  curr = prev = 1;      //curr は現在のFib値を保持する for
  (int i=3; i<=n; i++) { // 次の値を計算する
    past = prev;              // 過去はfibi(i-2)を保持する。
    prev = curr;              // prev は fibi(i-1) を保
    持 curr = past + prev;    //curr は現在 fibi(i) を保
    持
  }
  を返す;
}
```

関数Fibiは**for**ループを*n*-2回実行する。

(a) どちらのバージョンが理解しやすいですか? なぜですか?

(b) なぜFibrが**Fibiよりも**遅いのか説明してほしい。

2.12 ハノイの塔の一般化問題を解く再帰関数を書け。この問題では、小さいリングの上にリングが重ならない限り、各リングはどの極から始まってもよい。

2.13 セクション2.5のハノイの塔の再帰的実装を修正し、問題を解くのに必要な手のリストを返すようにする。

2.14 次の関数を考えてみよう:

```
static void foo (double val) {
  if (val != 0.0)
    foo(val/2.0);
```

```
}
```

この関数は再帰的に呼び出されるたびに基本ケースに向かって進む。理論上（つまり、もし**double**変数が真の実数のように振舞うなら）、この関数は0以外の数の入力に対して終了するのだろうか? 実際のコンピュータ実装では、この関数は終了するのだろうか?

2.15 *n個の*異なる整数値を含む配列の要素について、すべての順列を表示する関数を書きなさい。

2.16 最初の*n個の*集合のすべての部分集合を表示する再帰的アルゴリズムを書け。

正の整数。

2.17 2つの正の整数nと*mに対する*最大公約数(LCF)は、nと*m*の両方を均等に割る最大の整数である。LCF(*n*,*m*)は、*n*が*m* であると仮定すると、少なくとも1であり、最大でも*m* である。2,000年以上前、ユークリッドは、*n* mod *m* = 0のとき、LCF(*n*,*m*) = LCF(*m*, *n* mod *m*)であるという観察に基づいて、効率的なアルゴリズムを提供した。この事実を利用して、2つの正の整数のLCFを求める2つのアルゴリズムを書こう。最初のバージョンは反復的に値を計算する。もう1つは再帰を使って求めるものである。

2.18 素数の数が無限であることを矛盾によって証明せよ。

2.19 **(a)** 帰納法を用いて、$n^2 - n$ が常に偶数であることを示せ。

(b) $n^2 - n$ が常に偶数であることを、1文か2文で直接証明しなさい。

(c) $n^3 - n$は常に3で割り切れることを示せ。

(d) $n^5 - n$は5で割り切れるか? 答えを説明しなさい。

2.20 を証明する$\sum_{i=0}^{n-1} 2^i$ は不合理である。

2.21 その理由を

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} (n - i + 1) = \sum_{i=0}^{n-1} (n - i)$$

説明せよ。

2.22 数学的帰納法を用いて式2.2を証明せよ。

2.23 数学的帰納法を用いて式2.6を証明せよ。

2.24 数学的帰納法を用いて式2.7を証明せよ。

2.25 閉形式の解を求め、その解が和に対して正しいことを（帰納法を用いて）証明せよ。

$$\sum_{i=1}^{n} 3^i .$$

2.26 最初の*n個の*偶数の和が$n^2 + n$であることを証明せよ。

(a) 最初のn*個の*奇数の和が n^2 であると仮定する。

(b) 数学的帰納法によって。

2.27 a を整数とする和 $\sum_{i=a}^{n} i$ の閉形式を与えよ。

からn*の*間である。

2.28 $\mathrm{Fib}(n) < (\frac{5}{3})^n$ であることを証明せよ。

2.29 $n \geq 1$*のとき*、以下を証明せよ。 $\frac{n^2 (n+1)^2}{4}$.

$$\sum_{i=1}^{n} i^3 =$$

2.30 次の定理は**ピジョンホールの原理と**呼ばれる。

定理2.10 n + 1 *羽のハトが*n個の*穴をねぐらにするとき、少なくとも2羽のハトがいる穴が存在しなければならない。*

(a) 矛盾証明を使ってピジョンホールの原理を証明しなさい。

(b) 数学的帰納法を用いてピジョンホールの原理を証明せよ。

2.31 この問題では、3本以上の直線が1点で交わることがなく、2本の直線が平行にならないような、平面上の無限直線の配置を考える。

(a) *n 本の*線によって形成される領域の数を表す漸化式を与え、その漸化式が正しい理由を説明しなさい。

(b) 再帰を拡大した結果の合計を出す。

(c) 和の閉形式解を与えよ。

2.32 帰納法を用いて）漸化式 $\mathbf{T}(n) = \mathbf{T}(n - 1) + n; \mathbf{T}(1) = 1$ は、その閉形式解として$\mathbf{T}(n) = n(n + 1)/2$ を持つ。

2.33 次の漸化式を展開して閉形式の解を求め、帰納法を用いて答えが正しいことを証明しなさい。

$$\mathbf{T}(n) = 2\mathbf{T}(n - 1) + 1 \text{ for } n > 0; \mathbf{T}(0) = 0.$$

2.34 次の漸化式を展開して閉形式の解を求め、帰納法を用いて答えが正しいことを証明しなさい。

$$\mathbf{T}(n) = \mathbf{T}(n - 1) + 3n + 1 \text{ for } n > 0; \mathbf{T}(0) = 1.$$

2.35 *n*ビットの整数（標準的な2進数表記で表される）が、0から$2^n - 1$ の範囲内の任意の値を等確率で取ると仮定する。

(a) それぞれのビット位置について、その値が1になる確率と0になる確率は?

(b) n*ビットの*乱数の平均 "1 "ビット数は?

(c) 左端の "1 "ビットの位置の期待値は? 言い換えれば、"1 "ビットに遭遇するまでに、左から右へ移動する場合、平均していくつの位置を調べなければならないか。適切な和を示せ。

2.36 あなたの体の総容積は何リットル（ガロンでも可）ですか?

2.37 ある美術史家は、2万枚のフルスクリーンカラー画像のデータベースを持っている。

(a) どのくらいの容量が必要ですか? データベースを保存するのに必要なCDの枚数は? (CD1枚には約600MBのデータが入ります）。答えを導くために行ったすべての仮定を必ず説明してください。

(b) ここで、非圧縮画像に必要なスペースの1/10で画像を保存できる、優れた画像圧縮技術を利用できるとします。画像が圧縮されていれば、データベース全体が1枚のCDに収まるでしょうか?

2.38 ミシシッピ川河口から1日に流出する水の量は何立方マイルか？答えや補足的な事実を調べないでください。答えにたどり着くまでに行ったすべての仮定を必ず記述しなさい。

2.39 住宅ローンを購入する際、金利を安くするために事前にいくらかのお金（「ディスカウント・ポイント」と呼ばれる）を支払うという選択肢がよくある。ひとつは8%で前払い金なし、もうひとつは$7\frac{3}{4}$%で前払い金は住宅ローン額の1%である。低い方の金利で住宅ローンを借りた場合、1%の手数料を回収するのに何年かかるでしょうか？2つ目の、より正確な見積もりとして、高い方の金利を支払いながら、1%の手数料に相当する金額を5%の金利で銀行に投資していた場合、手数料と受け取るはずだった利息を回収するのにかかる時間はどのくらいでしょうか？この質問に答えるのに電卓は使わないでください。

2.40 家を新築する際、「建設ローン」という一時的な融資枠を借りることがある。建設期間終了後、建設ローンを通常の住宅ローンに切り替える。建設ローン期間中は、これまでに実際に借り入れた金額に対して請求された利息のみを毎月支払う。あなたの家の建設プロジェクトが4月の初めに始まり、6ヶ月後に完成すると仮定します。総工費は$300,000で、毎月初めに$50,000ずつ支払うと仮定します。建設ローンの利息は6%である。建設ローンの期間中に支払わなければならない利息の総額を見積もりなさい。

2.41 ここでは、高速コンピュータがどのように動作するかについて、あなたの知識を試す問題をいくつか出題します。ディスク・ドライブのアクセス時間は通常、ミリ秒（1000 分の 1 秒）かマイクロ秒（100 万分の 1 秒）か、どちらで測りますか？RAMメモリは1ワードを1マイクロ秒以上でアクセスしますか？マシンを常にフル回転させた場合、CPUは1年間にどれだけの命令を実行できますか？答えを導くのに紙や電卓を使わないでください。

2.42 あなたの家には100万ページ分の本がありますか? あなたの学校の図書館の建物には、全部で何ページありますか? どうしてその答えになったか説明してください。

2.43 この本にはいくつの単語が載っていますか? どうやって答えを出したか説明してください。

2.44 100万秒は何時間? 何日ですか? これらの質問に、頭の中ですべての計算をして答えなさい。どうやって答えを出したか説明しなさい。

2.45 アメリカにはいくつの市や町がありますか? その答えの根拠を説明しなさい。

2.46 ボストンからサンフランシスコまで歩くには何歩かかるか? 例題: あなたの答えがどのように得られたかを分かりやすく説明してください。

2.47 ある男性が義理の両親を訪ねるために車で旅を始めた。総走行距離は60マイルで、彼は時速60マイルで出発した。ちょうど1マイル走ったところで、彼は旅への熱意を失い、（瞬間的に）時速59マイルに減速した。さらに1マイル走ると、再び時速58マイルに減速する。これを1マイル進むごとに時速1マイルずつ減速しながら、旅を終えるまで続ける。

(a) 義理の両親のところに着くまで、男性はどれくらい時間がかかるのだろうか?

(b) 移動距離に応じて速度が滑らかに減少する連続的な場合、所要時間はどのくらいになるのだろうか?