Data Structure & Algorithms – Cấu Trúc Dữ Liệu & Giải Thuật

Nguyễn Quản Bá Hồng*

Ngày 7 tháng 7 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Data Structure & Algorithms – Cấu Trúc Dữ Liệu & Giải Thuật.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_structure_algorithm/NQBH_data_structure_algorithm.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_structure_algorithm/NQBH_data_structure_algorithm.tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

1 Basic Data Structure – Cấu Trúc Dữ Liệu Cơ Bản

2 Algorithms – Thuật Giải/Thuật Toán

2.1 [Cor+22]. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. 4e

[630 Amazon ratings][9071 Goodreads ratings]

- Amazon review. "A comprehensive update of the leading algorithms text, with new material on matchings in bipartite graphs, online algorithms, ML, & other topics. Some books on algorithms are rigorous but incomplete; others cover masses of material but lack rigor. *Introduction to Algorithms* uniquely combines rigor & comprehensiveness. It covers a broad range of algorithms in depth, yet makes their design & analysis accessible to all levels of readers, with self-contained chapters & algorithms in pseudocode. Since the publication of 1e, *Introduction to Algorithms* has become the leading algorithms text in universities worldwide as well as the standard reference for professionals. This 4e has been updated throughout. New for 4e:

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

- New chapters on matching in bipartite graphs, online algorithms, & ML
- New material on topics including solving recurrence equations, hash tables, potential functions, & suffix arrays
- 140 new exercises & 22 new problems
- Reader feedback-informed improvements to odd problems
- Clearer, more personal, & gender-neutral writing style
- Color added to improve visual presentation
- Notes, bibliography, & index updated to reflect developments in the field
- Website with new supplementary material

This book is a comprehensive update of the leading algorithms text, covering a broad range of algorithms in depth, yet making their design & analysis accessible to all levels of readers, with self-contained chapters & algorithms in pseudocode."

- "A data structure is a way to store & organize data in order to facilitate access & modifications."
- "Machine learning can be thought of as a method for performing algorithmic tasks without explicitly designing an algorithm, but instead inferring patterns from data & thereby automatically learning a solution."
- "The running time of an algorithm on a particular input is the number of instructions & data accesses executed."

About Author. Thomas H. Cormen is Emeritus Professor of Computer Science at Dartmouth College. Charles E. Leiserson is Edwin Sibley Webster Professor in Electrical Engineering & Computer Science at MIT. Ronald L. Rivest is Institute Professor at MIT. Clifford Stein is Wai T. Chang Professor of Industrial Engineering & Operations Research, & of Computer Science at Columbia University.

- Preface. Not so long ago, anyone who had heard word "algorithm" has almost certainly a computer scientist or mathematician. With computers having become prevalent in our modern lives, however, term is no longer esoteric (bí truyền). If look around home, find algorithms running in most mundane (tầm thường) places: microwave oven, washing machine, &, of course, computer. Ask algorithms to make recommendations to you: what music might like or what route to take when driving. Our society, for better or for worse, asks algorithms to suggest sentences for convicted criminals. Even rely on algorithms to keep you alive, or at least not to kill you: control systems in car or in medical equipment [To understand many of ways in which algorithms influence our daily lives, see book by Fry [162].]. Word "algorithm" appears somewhere in news seemingly every day.

  Therefore, it behooves you to understand algorithms not just as a student or practitioner of computer science, but as a citizen of world. Once understand algorithms, can educate others about what algorithms are, how they operate, & what their limitations are.

  This book provides a comprehensive introduction to modern study of computer algorithms. It presents many algorithms & covers them in considerable depth, yet makes their design accessible to all levels of readers. All analyses are laid out, some simple, some more involved. Have tried to keep explanations clear without sacrificing depth of coverage or mathematical rigor.

  Each chap presents an algorithm, a design technique, an application area, or a related topic. Algorithms are described in English & in a pseudocode designed to be readable by anyone who has done a little programming. Book contains 231 figures – many with multiple parts – illustrating how algorithms work. Since emphasize *efficiency* as a design criterion, include careful analyses of running times of algorithms.

  Text is intended primarily for use in undergraduate or graduate courses in algorithms or data structures. Because it discusses engineering issues in algorithm design, as well as mathematical aspects, it is equally well suited for self-study by technical professionals.

  In this, 4e, have once again updated entire book. Changes cover a broad spectrum, including new chaps & sects, color illustrations, & what hope you'll find to be a more engaging writing style.

  - To teacher. Have designed this book to be both versatile & complete. Should find it useful for a variety of courses, from an undergraduate course in data structures up through a graduate course in algorithms. Because have provided considerably more material than can fit in a typical 1-term course, can select material that best supports course wish to teach.

    Should find it easy to organize your course around just chaps you need. Have made chaps relatively self-contained, so that you need not worry about an unexpected & unnecessary dependence of 1 chap on another. Whereas in an undergraduate course, might use only some secs from a chap, in a graduate course, might cover entire chap.

    Have included 931 exercises & 162 problems. Each sect ends with exercises, & each chap ends with problems. Exercises are generally short questions that test basic mastery of material. Some are simple self-check thought exercises, but many are substantial & suitable as assigned homework. Problems include more elaborate case studies which often introduce new material. They often consist of several parts that lead student through steps required to arrive at a solution.

    As with 3e of this book, have made publicly available solutions to some, but by no means all, of problems & exercises. Can find these solutions on our website, http://mitpress.mit.edu/algorithms/. Want to check this site to see whether it contains solution to an exercise or problem that you plan to assign. Since set of solutions that post might grow over time, recommend: check site each time teach course.

Have starred sects & exercises that are more suitable for graduate students than for undergraduates. A starred sect is not necessarily more difficult than an unstarred one, but it may require an understanding of more advanced mathematics. Likewise, starred exercises may require an advanced background or more than average creativity.

- To student. Hope: this textbook provides you with an enjoyable introduction to field of algorithms. Have attempted to make every algorithm accessible & interesting. To help you when encounter unfamiliar or difficult algorithms, describe each one in a step-by-step manner. Also provide careful explanations of mathematics needed to understand analysis of algorithms & supporting figures to help visualize what is going on.

  Since this book is large, your class will probably cover only a portion of its material. Although hope: will find this book helpful to you as a course textbook now, have also tried to make it comprehensive enough to warrant space on your future professional bookshelf.

  What are prerequisites for reading this book?

  * Need some programming experience. In particular, should understand recursive procedures & simple data structures, e.g. arrays & linked lists (although Sect. 10.2 covers linked lists & a variant that you may find new).
  * You should have some facility with mathematical proofs, & especially proofs by mathematical induction. A few portions of book rely on some knowledge of elementary calculus. Although this book uses mathematics throughout, Part I & Appendices A–D teach you all mathematical techniques you will need.

  Website http://mitpress.mit.edu/algorithms/, links to solutions for some of problems & exercises. Feel free to check your solutions against ours.

- To the professional. Wide range of topics in this book makes it an excellent handbook on algorithms. Because each chap is relatively self-contained, can focus on topics most relevant to you.

  Since most of algorithms discuss have great practical utility, address implementation concerns & other engineering issues. Often provide practical alternatives to few algorithms that are primarily of theoretical interest.

  If wish to implement any of algorithms, should find translation of our pseudocode into your favorite programming language to be a fairly straightforward task. Have designed pseudocode to present each algorithm clearly & succinctly. Consequently, do not address error handling & other software-engineering issues that require specific assumptions about your programming environment. Attempt to present each algorithm simply & directly without allowing idiosyncrasies of a particular programming language to obscure its essence. If used to 0-origin arrays, might find our frequent practice of indexing arrays from 1 a minor stumbling block. Can always either subtract 1 from our indices or just overallocate array & leave position 0 unused.

  Understand: if using this book outside of a course, then might be unable to check your solutions to problems & exercises against solutions provided by an instructor. Website http://mitpress.mit.edu/algorithms/, links to solutions for some of problems & exercises so that can check work.

- To colleagues. Have supplied an extensive bibliography & pointers to current literature. Each chap ends with a set of chap notes that give historical details & references. Chap notes do not provide a complete reference to whole field of algorithms, however. Though it may be hard to believe for a book of this size, space constraints prevented us from including many interesting algorithms.

  Despite myriad requests from students for solutions to problems & exercises, have adopted policy of not citing references for them, removing temptation for students to look up a solution rather than to discover it themselves.

- Changes for 4e. As said about changes for 2e & 3e, depending on how you look at it, book changed either not much or quite a bit. A quick look at table of contents shows: most of 3e chaps & sects appear in 4e. Removed 3 chaps & several sects, but have added 3 new chaps & several new sects apart from these new chaps.

  Kept hybrid organization (tổ chức hỗn hợp) from 1st 3 editions. Rather than organizing chaps only by problem domains or only according to techniques, this book incorporates elements of both. It contains technique-based chaps on divide-&-conquer, dynamic programming, greedy algorithms, amortized analysis, augmenting data structures, NP-completeness, & approximation algorithms. But it also has entire parts on sorting, on data structures for dynamic sets, & on algorithms for graph problems. Find: although need to know how to apply techniques for designing & analyzing algorithms, problems seldom announce to you which techniques are most amenable to solving them.

  Some of changes in 4e apply generally across book, & some are specific to particular chaps or sects. Here is a summary of most significant general changes:

  * Added 140 new exercises & 22 new problems. Also improved many of old exercises & problems, often as result of reader feedback. (Thanks to all readers who made suggestions.)
  * Have color! With designers from MIT Press, selected a limited palette, devised to convey information & to be pleasing to eye. (Delighted to display red-black trees in – get this – red & black!) To enhance readability, defined terms, pseudocode comments, & page numbers in index are in color.
  * Pseudocode procedures appear on a tan background to make them easier to spot, & they do not necessarily appear on page of their 1st ref. When they don't, text directs you to relevant page. In same vein, nonlocal refs to numbers equations, theorems, lemmas, & corollaries include page number.
  * Removed topics that were rarely taught. Dropped in their entirety chaps on Fibonacci heaps, van Emde Boas trees, & computational geometry. In addition, following material was excised (vật liệu đã được cắt bỏ): maximum-subarray problem, implementing pointers & objects, perfect hashing, randomly built binary search trees, matroids, push-relabel algorithms for maximum flow, iterative fast Fourier transform method, details of simplex algorithm for linear programming, & integer factorization. Can find all removed material on website http://mitpress.mit.edu/algorithms/.

* Reviewed entire book & rewrote sentences, paragraphs, & sects to make writing cleaner, more personal, & gender neural. E.g., "traveling-salesman problem" in prev editions is now called "traveling-salesperson problem." Believe: critically important for engineering & science, including our own field of CS, to be welcoming to everyone. (The 1 place that stumped us is in Chap. 13, which requires a term for a parent's sibling. Because English language has no such gender-neural term, regretfully stuck with "uncle".)
* Chap notes, bibliography, & index were updated, reflecting dramatic growth of field of algorithsm since 3e.
* Corrected errors, posting most corrections on our website of 3e errata. Those what were reported while were in full swing preparing this edition were not posted, but were corrected in this edition. (Thanks again to all readers who helped us identify issues.)

Specific changes for 4e include following:

* Renamed Chap. 3 & added a sect giving an overview of asymptotic notation before delving into formal defs.
* Chap. 4 underwent substantial changes to improve its mathematical foundation & make it more robust & intuitive. Notion of an algorithmic recurrence was introduced, & topic of ignoring floors & ceilings in recurrences was addressed more rigorously. 2nd case of master theorem incorporates polylogarithmic factors, & a rigorous proof of a "continuous" version of master theorem is now provided. Also present powerful & general Akra-Bazzi methodd (without proof).
* Deterministic order-statistic algorithm in Chap. 9 is slightly different, & analyses of both randomized & deterministic order-statistic algorithms have been revamped.
* In addition to stacks & queues, Sect. 10.1 discusses ways to store arrays & matrices.
* Chap. 11 on hash tables includes a modern treatment of hash functions. Also emphasize linear probing as an efficient method for resolving collisions when underlying hardware implements caching to favor local searches.
* To replace sects on matroids in Chap. 15, converted a problem in 3e about offline caching into a full sect.
* Sect. 16.4 now contains a more intuitive explanation of potential functions to analyze table doubling & halving.
* Chap. 17 on augmenting data structures was relocated from Part III to Part V, reflecting our view: this techniques goes beyond basic material.
* Chap. 25 is a new chap about matchings in bipartite graphs. It presents algorithms to find a matching of maximum cardinality, to solve stable-marriage problem, & to find a maximum-weight matching (known as "assignment problem").
* Chap. 26, on task-parallel computing, has been updated with modern terminology, including name of chap.
* Chap. 27, which covers online algorithms, is another new chap. In an online algorithm, input arrives over time, rather than being available in its entirety at start of algorithm. Chap describes several examples of online algorithms, including determining how long to wait for an elevator before taking stairs, maintaining a linked list via move-to-front heuristic, & evaluating replacement policies for caches.
* In Chap. 29, removed detailed representation of simplex algorithm, as it was math heavy without really conveying many algorithmic ideas. Chap now focuses on key aspect of how to model problems as linear programs, along with essential duality property of linear programming.
* Sect. 32.5 adds to chap on string matching simple, yet powerful, structure of suffix arrays.
* Chap. 33, on ML, is 3rd new chap. Introduce several basic methods used in ML: clustering to group similar items together, weighted-majority algorithms, & gradient descent to find minimizer of a function.
* Sect. 34.5.6 summarizes strategies for polynomial-time reductions to show: problems are NP-hard.
* Proof of approximation algorithm for set-covering problem in Sect. 35.3 has been revised.

- Website. Can use website http://mitpress.mit.edu/algorithms/, to obtain supplementary information & to communicate with us. Website links to a list of known errors, material from 3e that is not included in 4e, solutions to selected exercises & problems, Python implementations of many of algorithms in this book, a list explaining corny professor jokes (of course), as well as other content, which may add to. Website also tells how to report errors or make suggestions.

• I. FOUNDATIONS.

- Introduction. When design & analyze algorithms, need to be able to describe how they operate & how to design them. Also need some mathematical tools to show: your algorithms do right thing & do it efficiently. This part will get you started. Later parts of this book will build upon this base.

1. Chap. 1 provides an overview of algorithms & their place in modern computing systems. This chap defines what an algorithm is & lists some examples. It also makes a case for considering algorithms as a technology, alongside technologies e.g. fast hardware, graphical user interfaces, object-oriented systems, & networks.
2. Chap. 2, see 1st algorithms, which solve problem of sorting a sequence of n numbers. They are written in a pseudocode which, although not directly translatable to any conventional programming language, conveys structure of algorithm clearly enough that you should be able to implement it in language of your choice. Sorting algorithms examined are insertion sort, which uses an incremental approach, & merge sort, which uses a recursive technique known as "divide-&-conquer." Although time each requires increases with value of n, rate of increase differs between 2 algorithms. Determine these running times in Chap. 2 & develop a useful "asymptotic" notation to express them.
3. Chap. 3 precisely defines asymptotic notation. Use asymptotic notation to bound growth of functions – most often, functions that describe running time of algorithms – from above & below. Chap starts by informally defining most

commonly used asymptotic notations & giving an example of how to apply them. It then formally defines 5 asymptotic notations & presents conventions for how to put them together. Rest of Chap. 3 is primarily a presentation of mathematical notation, more to ensure your use of notation matches that in this book than to teach new mathematical concepts.

4. Chap. 4 delves further into divide-&-conquer method introduced in Chap. 2. It provides 2 additional examples of divide-&-conquer algorithms for multiplying square matrices, including Strassen's surprising method. Chap. 4 contains methods for solving recurrences, which are useful for describing running times of recursive algorithms. In substitution method, guess an answer & prove it correct. Recursion trees provide 1 way to generate a guess. Chap. 4 also presents powerful technique of "master method", which can often use to solve recurrences that arise from divide-&-conquer algorithms. Although chap provides a proof of a foundational theorem on which master theorem depends, should feel free to employ master method without delving into proof. Chap. 4 concludes with some advanced topics.
5. Chap. 5 introduces probabilistic analysis & randomized algorithms. Typically use probabilistic analysis to determine running time of an algorithm in cases in which, due to presence of an inherent probability distribution, running time may differ on different inputs of same size. In some cases, might assume: inputs conform to a known probability distribution, so that you are averaging running time over all possible inputs. In other cases, probability distribution comes not from inputs but from random choices made during course of algorithm. An algorithm whose behavior is determined not only by its input but by values produced by a random-number generator is a randomized algorithm. Can use randomized algorithms to enforce a probability distribution on inputs – thereby ensuring: no particular input always causes poor performance – or even to bound error rate of algorithms that are allowed to produce incorrect results on a limited basis.
6. Appendices A–D contain other mathematical material that you will find helpful as read this book. Might have seen much of material in appendix chaps before having read this book (although specific defs & notational conventions use may differ in some cases from what you have seen in past), & so you should think of appendices as reference material. On other hand, probably have not already seen most of material in Part I. All chaps in Part I & appendices are written with a tutorial flavor.

◦ 1. Role of Algorithms in Computing. What are algorithms? Why is study of algorithms worthwhile? What is role of algorithms relative to other technologies used in computers? This chap will answer these questions.

– Vai trò của thuật toán trong máy tính. Thuật toán là gì? Tại sao việc nghiên cứu thuật toán lại có giá trị? Vai trò của thuật toán so với các công nghệ khác được sử dụng trong máy tính là gì? Chương này sẽ trả lời những câu hỏi này.

∗ 1.1. Algorithms. Informally, an *algorithm* is any well-defined computational procedure that takes some value, or set of values, as input & produces some value, or set of values, as output in a finite amount of time. An algorithm is thus a sequence of computational steps that transform input into output.

– Theo nghĩa thông thường, thuật toán là bất kỳ quy trình tính toán được xác định rõ ràng nào lấy 1 số giá trị hoặc tập hợp các giá trị làm đầu vào & tạo ra 1 số giá trị hoặc tập hợp các giá trị làm đầu ra trong 1 khoảng thời gian hữu hạn. Do đó, thuật toán là 1 chuỗi các bước tính toán biến đổi đầu vào thành đầu ra.

Can also view an algorithm as a tool for solving a well-specified computational problem. Statement of problem specifies in general terms desired input/output relationship for problem instances, typically of arbitrarily large size. Algorithm describes a specific computational procedure for achieving that input/output relationship for all problem instances.

– Cũng có thể xem thuật toán như 1 công cụ để giải quyết 1 vấn đề tính toán được chỉ định rõ ràng. Phát biểu vấn đề chỉ định theo các thuật ngữ chung mối quan hệ đầu vào/đầu ra mong muốn cho các trường hợp vấn đề, thường có kích thước lớn tùy ý. Thuật toán mô tả 1 quy trình tính toán cụ thể để đạt được mối quan hệ đầu vào/đầu ra đó cho tất cả các trường hợp vấn đề.

E.g., suppose need to sort a sequence of numbers into monotonically increasing order. This problem arises frequently in practice & provides fertile ground for introducing many standard design techniques & analysis tools. Here is how we formally define *sorting problem*: Input: A sequence of n numbers $a_1, \ldots, a_n$. Output: A permutation (reordering) $a'_1, \ldots, a'_n$ of input sequence s.t. $a'_1 \leq \cdots \leq a'_n$. Thus, given input sequence ..., a correct sorting algorithm returns as output sequence .... Such an input sequence is called an instance of sorting problem. In general, an instance of a problem consists of input (satisfying whatever constraints are imposed in problem statement) needed to compute a solution to problem.

– E.g., giả sử cần sắp xếp 1 dãy số theo thứ tự tăng đơn điệu. Bài toán này thường phát sinh trong thực tế & cung cấp nền tảng màu mỡ để giới thiệu nhiều kỹ thuật thiết kế chuẩn & công cụ phân tích. Sau đây là cách chúng tôi định nghĩa chính thức bài toán sắp xếp: Đầu vào: Một dãy n số $a_1, \ldots, a_n$. Đầu ra: Một hoán vị (sắp xếp lại) $a'_1, \ldots, a'_n$ của dãy đầu vào s.t. $a'_1 \leq \cdots \leq a'_n$. Do đó, với dãy đầu vào ..., 1 thuật toán sắp xếp đúng sẽ trả về dãy đầu ra .... Một dãy đầu vào như vậy được gọi là 1 trường hợp của bài toán sắp xếp. Nhìn chung, 1 trường hợp của bài toán bao gồm đầu vào (thỏa mãn bất kỳ ràng buộc nào được áp dụng trong phát biểu bài toán) cần thiết để tính toán giải pháp cho bài toán.

Because many problems use it as an intermediate step, sorting is a fundamental operation in CS. As a result, you have a large number of good sorting algorithms at your disposal. Which algorithm is best for a given application depends on – among other factors – number of items to be sorted, extent to which items are already somewhat sorted, possible restrictions on item values, architecture of computer, & kind of storage devices to be used: main memory, disks, or even – archaically – tapes.

– Vì nhiều bài toán sử dụng nó như 1 bước trung gian, nên sắp xếp là 1 hoạt động cơ bản trong CS. Do đó, bạn có 1 số lượng lớn các thuật toán sắp xếp tốt theo ý mình. Thuật toán nào là tốt nhất cho 1 ứng dụng nhất định phụ thuộc vào – trong số các yếu tố khác – số lượng mục cần sắp xếp, mức độ các mục đã được sắp xếp phần nào, các hạn chế có thể có đối với giá trị mục, kiến trúc máy tính, & loại thiết bị lưu trữ sẽ được sử dụng: bộ nhớ chính, đĩa hoặc thậm chí – theo cách cổ xưa – băng.

An algorithm for a computational problem is correct if, for every problem instance provided as input, it halts – finishes its computing in finite time – & outputs correct solution to problem instance. A correct algorithm solves given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if you can control their error rate. See an example of an algorithm with a controllable error rate in Chap. 31 when study algorithms for finding large prime numbers. Ordinarily, however, concern only with correct algorithms.

– 1 thuật toán cho 1 bài toán tính toán là đúng nếu, đối với mọi trường hợp bài toán được cung cấp làm đầu vào, nó dừng lại – hoàn tất việc tính toán trong thời gian hữu hạn – & đưa ra giải pháp đúng cho trường hợp bài toán. Một thuật toán đúng giải quyết được bài toán tính toán đã cho. Một thuật toán sai có thể không dừng lại ở 1 số trường hợp đầu vào hoặc có thể dừng lại với 1 câu trả lời sai. Trái ngược với những gì bạn có thể mong đợi, đôi khi các thuật toán sai có thể hữu ích, nếu bạn có thể kiểm soát tỷ lệ lỗi của chúng. Xem ví dụ về 1 thuật toán có tỷ lệ lỗi có thể kiểm soát được trong Chương 31 khi nghiên cứu các thuật toán để tìm số nguyên tố lớn. Tuy nhiên, thông thường chỉ quan tâm đến các thuật toán đúng.

An algorithm can be specified in English, as a computer program, or even as a hardware design. Only requirement: specification must provide a precise description of computational procedure to be followed.

– Thuật toán có thể được chỉ định bằng tiếng Anh, dưới dạng chương trình máy tính hoặc thậm chí là thiết kế phần cứng. Yêu cầu duy nhất: đặc tả phải cung cấp mô tả chính xác về quy trình tính toán cần tuân theo.

What kinds of problems are solved by algorithms? Sorting is by o means only computational problem for which algorithms have been developed. Practical applications of algorithms are ubiquitous & including following examples:

– **Các loại vấn đề nào được giải quyết bằng thuật toán?** Sắp xếp theo nghĩa là chỉ có vấn đề tính toán mà thuật toán được phát triển. Các ứng dụng thực tế của thuật toán có ở khắp mọi nơi & bao gồm các ví dụ sau:

· Human Genome Project has made great progress toward goals of identifying all roughly 30000 genes in human DNA< determining sequences of roughly 3 billion chemical base pairs that make up human DNA, storing this information in databases, & developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although solutions to various problems involved are beyond scope of this book, many methods to solve these biological problems use ideas presented here, enabling scientists to accomplish tasks while using resources efficiently. Dynamic programming, as in Chap. 14, is an important technique for solving several of these biological problems, particularly ones that involve determining similarity between DNA sequences. Savings realized are in time, both human & machine, & in money, as more information can be extracted by laboratory techniques.

  – Dự án bộ gen người đã đạt được tiến bộ lớn hướng tới mục tiêu xác định tất cả khoảng 30000 gen trong DNA của con người, xác định trình tự của khoảng 3 tỷ cặp bazơ hóa học tạo nên DNA của con người, lưu trữ thông tin này trong cơ sở dữ liệu, & phát triển các công cụ để phân tích dữ liệu. Mỗi bước trong số này đều yêu cầu các thuật toán phức tạp. Mặc dù các giải pháp cho nhiều vấn đề liên quan nằm ngoài phạm vi của cuốn sách này, nhưng nhiều phương pháp để giải quyết các vấn đề sinh học này sử dụng các ý tưởng được trình bày ở đây, cho phép các nhà khoa học hoàn thành nhiệm vụ trong khi sử dụng tài nguyên 1 cách hiệu quả. Lập trình động, như trong Chương 14, là 1 kỹ thuật quan trọng để giải quyết 1 số vấn đề sinh học này, đặc biệt là những vấn đề liên quan đến việc xác định điểm tương đồng giữa các trình tự DNA. Tiết kiệm được về mặt thời gian, cả con người & máy móc, & tiền bạc, vì có thể trích xuất nhiều thông tin hơn bằng các kỹ thuật trong phòng thí nghiệm.

· Internet enables people all around world to quickly access & retrieve large amounts of information. With aid of clever algorithms, sites on internet are able to manage & manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which data travels (techniques for solving such problems appear in Chap. 22), & using a search engine to quickly find pages on which particular information resides (related techniques are in Chaps. 11 & 32).

  – Internet cho phép mọi người trên toàn thế giới truy cập nhanh & lấy thông tin với số lượng lớn. Với sự trợ giúp của các thuật toán thông minh, các trang web trên internet có thể quản lý & thao tác với khối lượng dữ liệu lớn này. Các ví dụ về các vấn đề sử dụng thuật toán thiết yếu bao gồm tìm các tuyến đường tốt mà dữ liệu di chuyển (các kỹ thuật để giải quyết các vấn đề như vậy xuất hiện trong Chương 22), & sử dụng công cụ tìm kiếm để nhanh chóng tìm các trang có thông tin cụ thể (các kỹ thuật liên quan có trong Chương 11 & 32).

· Electronic commerce enables goods & services to be negotiated & exchanged electronically, & it depends on privacy of personal information e.g. credit card numbers, passwords, & bank statements. Core technologies used in electronic commerce include public-key cryptography & digital signatures (covered in Chap. 31), which are based on numerical algorithms & number theory.

  – Thương mại điện tử cho phép hàng hóa & dịch vụ được thương lượng & trao đổi điện tử, & phụ thuộc vào quyền riêng tư của thông tin cá nhân, ví dụ như số thẻ tín dụng, mật khẩu, & sao kê ngân hàng. Các công nghệ cốt lõi được sử dụng trong thương mại điện tử bao gồm mật mã khóa công khai & chữ ký số (được đề cập trong Chương 31), dựa trên các thuật toán số & lý thuyết số.

· Manufacturing & other commercial enterprises often need to allocate scarce resources in most beneficial way. An oil company might wish to know where to place its wells in order to maximize its expected profit. A political candidate might want to determine where to spend money buying campaign advertising in order to maximize chances of winning an election. An airline might wish to assign crews to flights in least expensive way possible, making sure that each flight is covered & government regulations regarding crew scheduling are met. An internet service provider might wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved by modeling them as linear programs, which Chap. 29 explores.

– Sản xuất & các doanh nghiệp thương mại khác thường cần phân bổ các nguồn lực khan hiếm theo cách có lợi nhất. Một công ty dầu mỏ có thể muốn biết vị trí đặt giếng dầu của mình để tối đa hóa lợi nhuận dự kiến. Một ứng cử viên chính trị có thể muốn xác định nơi chi tiền mua quảng cáo chiến dịch để tối đa hóa cơ hội chiến thắng trong cuộc bầu cử. Một hãng hàng không có thể muốn phân công phi hành đoàn cho các chuyến bay theo cách ít tốn kém nhất có thể, đảm bảo rằng mỗi chuyến bay đều được bảo hiểm & các quy định của chính phủ về lịch trình của phi hành đoàn được đáp ứng. Một nhà cung cấp dịch vụ internet có thể muốn xác định vị trí đặt thêm các nguồn lực để phục vụ khách hàng hiệu quả hơn. Tất cả những điều này là ví dụ về các vấn đề có thể được giải quyết bằng cách mô hình hóa chúng dưới dạng các chương trình tuyến tính, mà Chương 29 sẽ khám phá.

Although some of details of these examples are beyond scope of this book, do give underlying techniques that apply to these problems & problem areas. Also show how to solve many specific problems, including following:

– Mặc dù 1 số chi tiết của các ví dụ này nằm ngoài phạm vi của cuốn sách này, nhưng hãy đưa ra các kỹ thuật cơ bản áp dụng cho các vấn đề này & các lĩnh vực có vấn đề. Cũng chỉ ra cách giải quyết nhiều vấn đề cụ thể, bao gồm:

· You have a road map on which distance between each pair of adjacent intersection is marked, & you wish to determine shortest route from 1 intersection to another. Number of possible routes can be huge, even if you disallow routes that cross over themselves. How can you choose which of all possible routes is shortest? Can start by modeling road map (which is itself a model of actual roads) as a graph (meet in Part VI & Appendix B). In this graph, wish to find shortest path from 1 vertex to another. Chap. 22 shows how to solve this problem efficiently.

  – Bạn có 1 bản đồ đường bộ trên đó khoảng cách giữa mỗi cặp giao lộ liền kề được đánh dấu, & bạn muốn xác định tuyến đường ngắn nhất từ 1giao lộ đến giao lộ khác. Số lượng các tuyến đường khả thi có thể rất lớn, ngay cả khi bạn không cho phép các tuyến đường cắt nhau. Làm thế nào bạn có thể chọn tuyến đường nào trong số tất cả các tuyến đường khả thi là ngắn nhất? Có thể bắt đầu bằng cách mô hình hóa bản đồ đường bộ (bản thân nó là 1 mô hình của các con đường thực tế) dưới dạng đồ thị (gặp trong Phần VI & Phụ lục B). Trong đồ thị này, muốn tìm đường đi ngắn nhất từ 1 đỉnh đến đỉnh khác. Chương 22 cho thấy cách giải quyết vấn đề này 1 cách hiệu quả.

· Given a mechanical design in terms of a library of parts, where each part may include instances of other parts, list parts in order so that each part appears before any part that uses it. If design comprises n parts, then there are $n!$ possible orders. Because factorial function grows faster than even an exponential function, you cannot feasibly generate each possible order & then verify: within that order, each part appears before parts using it (unless you have only a few parts). This problem is an instance of topological sorting, & Chap. 20 shows how to solve this problem efficiently.

  – Với 1 thiết kế cơ khí theo dạng thư viện các bộ phận, trong đó mỗi bộ phận có thể bao gồm các trường hợp của các bộ phận khác, hãy liệt kê các bộ phận theo thứ tự sao cho mỗi bộ phận xuất hiện trước bất kỳ bộ phận nào sử dụng nó. Nếu thiết kế bao gồm n bộ phận, thì có $n!$ thứ tự khả thi. Vì hàm giai thừa tăng nhanh hơn cả hàm mũ, nên bạn không thể tạo ra từng thứ tự khả thi & sau đó xác minh: trong thứ tự đó, mỗi bộ phận xuất hiện trước các bộ phận sử dụng nó (trừ khi bạn chỉ có 1 vài bộ phận). Bài toán này là 1 ví dụ về sắp xếp tôpô, & Chương 20 cho thấy cách giải quyết vấn đề này 1 cách hiệu quả.

· You need to compress a large file containing text so that it occupies less space. Many ways to do so are known, including "LZW compression", which looks for repeating character sequences. Chap. 15 studies a different approach, "Huffman coding," which encodes characters by bit sequences of various lengths, with characters occurring more frequently encoded by shorter bit sequences.

  – Bạn cần nén 1 tệp lớn chứa văn bản để nó chiếm ít không gian hơn. Có nhiều cách để thực hiện việc này, bao gồm "nén LZW", tìm kiếm các chuỗi ký tự lặp lại. Chương 15 nghiên cứu 1 cách tiếp cận khác, "mã hóa Huffman", mã hóa các ký tự theo chuỗi bit có độ dài khác nhau, với các ký tự xuất hiện thường xuyên hơn được mã hóa bằng chuỗi bit ngắn hơn.

These lists are far from exhaustive (as you again have probably surmised from this book's heft), but they exhibit 2 characteristics common to many interesting algorithmic problems:

– Những danh sách này còn lâu mới đầy đủ (như bạn có thể đã đoán ra từ độ dày của cuốn sách này), nhưng chúng thể hiện 2 đặc điểm chung của nhiều bài toán thuật toán thú vị:

1. They have many candidate solutions, overwhelming majority of which do not solve problem at hand. Finding one that does, or one that is "best," without explicitly examining each possible solution, can present quite a challenge.

   – Họ có nhiều giải pháp ứng viên, phần lớn trong số đó không giải quyết được vấn đề hiện tại. Việc tìm ra giải pháp có thể giải quyết được vấn đề hoặc giải pháp "tốt nhất" mà không xem xét rõ ràng từng giải pháp khả thi có thể là 1 thách thức lớn.

2. They have practical applications. Of problems in above list, finding shortest path provides easiest examples. A transportation firm, e.g. a trucking or railroad company, has a financial interest in finding shortest paths through a road or rail network because taking shorter paths results in lower labor & fuel costs. Or a routing node on internet might need to find shortest path through network in order to route a message quickly. Or a person wishing to drive from New York to Boston might want to find driving directions using a navigation app.

   – Chúng có ứng dụng thực tế. Trong số các vấn đề trong danh sách trên, tìm đường đi ngắn nhất cung cấp các ví dụ dễ nhất. Một công ty vận tải, ví dụ như công ty xe tải hoặc đường sắt, có lợi ích tài chính trong việc tìm đường đi ngắn nhất qua mạng lưới đường bộ hoặc đường sắt vì đi theo đường ngắn hơn sẽ tiết kiệm chi phí lao động & nhiên liệu. Hoặc 1 nút định tuyến trên internet có thể cần tìm đường đi ngắn nhất qua mạng lưới để định tuyến tin nhắn nhanh chóng. Hoặc 1 người muốn lái xe từ New York đến Boston có thể muốn tìm chỉ đường lái xe bằng ứng dụng dẫn đường.

Not every problem solved by algorithms has an easily identified set of candidate solutions. E.g., given a set of numerical

values representing samples of a signal taken at regular time intervals, discrete Fourier transform converts time domain to frequency domain. I.e., it approximates signal as a weighted sum of sinusoids, producing strength of various frequencies which, when summed, approximate sampled signal. In addition to lying at heart of signal processing, discrete Fourier transforms have applications in data compression & multiplying large polynomials & integers. Chap. 30 gives an efficient algorithm, fast Fourier transform (commonly called FFT), for this problem. Chap also sketches out design of a hardware FFT circuit.

– Không phải mọi vấn đề được giải quyết bằng thuật toán đều có 1 tập hợp các giải pháp ứng viên dễ dàng xác định. Ví dụ, với 1 tập hợp các giá trị số biểu diễn các mẫu tín hiệu được lấy theo các khoảng thời gian đều đặn, phép biến đổi Fourier rời rạc chuyển đổi miền thời gian sang miền tần số. Tức là, nó xấp xỉ tín hiệu như 1 tổng có trọng số của các sin, tạo ra cường độ của các tần số khác nhau, khi được cộng lại, sẽ xấp xỉ tín hiệu được lấy mẫu. Ngoài việc nằm ở trung tâm của xử lý tín hiệu, phép biến đổi Fourier rời rạc còn có ứng dụng trong nén dữ liệu & nhân các đa thức lớn & số nguyên. Chương 30 đưa ra 1 thuật toán hiệu quả, phép biến đổi Fourier nhanh (thường gọi là FFT), cho bài toán này. Chương này cũng phác thảo thiết kế của 1 mạch FFT phần cứng.

Data structures. This book also presents several data structures. A *data structure* is a way to store & organize data in order to facilitate access & modifications. Using appropriate data structure or structures is an important part of algorithm design. No single data structure works well for all purposes, & so you should know strengths & limitations of serval of them.

– **Cấu trúc dữ liệu.** Cuốn sách này cũng trình bày 1 số cấu trúc dữ liệu. *cấu trúc dữ liệu* là 1 cách để lưu trữ & tổ chức dữ liệu nhằm tạo điều kiện truy cập & sửa đổi. Sử dụng cấu trúc dữ liệu hoặc các cấu trúc phù hợp là 1 phần quan trọng của thiết kế thuật toán. Không có cấu trúc dữ liệu đơn lẻ nào hoạt động tốt cho mọi mục đích, & vì vậy bạn nên biết điểm mạnh & hạn chế của 1 số cấu trúc.

∗ **Technique.** Although can use this book as a "cookbook" for algorithms, might someday encounter a problem for which you cannot readily find a published algorithm (e.g., many of exercises & problems in this book). This book will teach you techniques of algorithm design & analysis so that you can develop algorithms on your own, show that they give correct answer, & analyze their efficiency. Different chaps address different aspects of algorithmic problem solving. Some chaps address specific problems, e.g. finding medians & order statistics in Chap. 9, computing minimum spanning trees in Chap. 21, & determining a maximum flow in a network in Chap. 24. Other chaps introduce techniques, e.g. divide-&-conquer in Chaps. 2 & 4, dynamic programming in Chap. 14, & amortized analysis in Chap. 16.

– **Kỹ thuật.** Mặc dù có thể sử dụng cuốn sách này như 1 "sách hướng dẫn" về thuật toán, nhưng 1 ngày nào đó bạn có thể gặp phải 1 vấn đề mà bạn không dễ dàng tìm thấy thuật toán đã công bố (ví dụ: nhiều bài tập & bài toán trong cuốn sách này). Cuốn sách này sẽ dạy bạn các kỹ thuật thiết kế & phân tích thuật toán để bạn có thể tự phát triển thuật toán, chứng minh rằng chúng đưa ra câu trả lời đúng, & phân tích hiệu quả của chúng. Các chương khác nhau giải quyết các khía cạnh khác nhau của việc giải quyết vấn đề thuật toán. Một số chương giải quyết các vấn đề cụ thể, ví dụ: tìm trung vị & thống kê thứ tự trong Chương 9, tính toán cây khung nhỏ nhất trong Chương 21, & xác định luồng cực đại trong mạng trong Chương 24. Các chương khác giới thiệu các kỹ thuật, ví dụ: chia-&-trị trong Chương 2 & 4, lập trình động trong Chương 14, & phân tích khấu hao trong Chương 16.

∗ **Hard problems.** Most of this book is about efficient algorithms. Our usual measure of efficiency is speed: how long does an algorithm take to produce its result? There are some problems, however, for which we know of no algorithm that runs in a reasonable amount of time. Chap. 34 studies an interesting subset of these problems, which are known as NP-complete.

– **Các bài toán khó.** Phần lớn cuốn sách này nói về các thuật toán hiệu quả. Thước đo hiệu quả thông thường của chúng ta là tốc độ: 1 thuật toán mất bao lâu để tạo ra kết quả của nó? Tuy nhiên, có 1 số bài toán mà chúng ta không biết thuật toán nào chạy trong khoảng thời gian hợp lý. Chương 34 nghiên cứu 1 tập hợp con thú vị của các bài toán này, được gọi là NP-complete.

Why are NP-complete problems interesting? 1st, although no efficient algorithm for an NP-complete has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. I.e., no one knows whether efficient algorithms exist for NP-complete problems. 2nd, set of NP-complete problems has remarkable property that if an efficient algorithm exists for any 1 of them, then efficient algorithms exist for all of them. This relationship among NP-complete problems make lack of efficient solutions all more tantalizing. 3rd, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Computer scientists are intrigued by how a small change to problem statement can cause a big change to efficiency of best known algorithm.

– Tại sao các bài toán NP-complete lại thú vị? Thứ nhất, mặc dù chưa từng tìm thấy thuật toán hiệu quả nào cho 1 NP-complete, nhưng chưa từng có ai chứng minh được rằng không thể tồn tại thuật toán hiệu quả cho 1 NP-complete. Nghĩa là không ai biết liệu các thuật toán hiệu quả có tồn tại cho các bài toán NP-complete hay không. Thứ hai, tập hợp các bài toán NP-complete có tính chất đáng chú ý là nếu 1 thuật toán hiệu quả tồn tại cho bất kỳ 1 bài toán nào trong số chúng, thì các thuật toán hiệu quả tồn tại cho tất cả các bài toán đó. Mối quan hệ này giữa các bài toán NP-complete khiến cho việc thiếu các giải pháp hiệu quả trở nên hấp dẫn hơn. Thứ ba, 1 số bài toán NP-complete tương tự, nhưng không giống hệt, với các bài toán mà chúng ta biết về các thuật toán hiệu quả. Các nhà khoa học máy tính rất thích thú với cách mà 1 thay đổi nhỏ trong phát biểu bài toán có thể gây ra sự thay đổi lớn về hiệu quả của thuật toán được biết đến nhiều nhất.

You should know about NP-complete problems because some of them arise surprisingly often in real applications. If you are called upon to produce an efficient algorithm for an NP-complete problem, you are likely to spend a lot of time in a fruitless search. If, instead, you can show: problem is NP-complete, you can spend your time developing an efficient

approximation algorithm, i.e., an algorithm that gives a good, but not necessarily best possible, solution.
– Bạn nên biết về các bài toán NP-complete vì 1 số bài toán này thường xuất hiện trong các ứng dụng thực tế. Nếu bạn được yêu cầu tạo ra 1 thuật toán hiệu quả cho 1 bài toán NP-complete, bạn có thể sẽ dành nhiều thời gian vào việc tìm kiếm vô ích. Nếu thay vào đó, bạn có thể chỉ ra: bài toán là NP-complete, bạn có thể dành thời gian để phát triển 1 thuật toán xấp xỉ hiệu quả, tức là 1 thuật toán đưa ra 1 giải pháp tốt, nhưng không nhất thiết là giải pháp tốt nhất có thể.
As a concrete example, consider a delivery company with a central depot. Each day, it loads up delivery trucks at depot & sends them around to deliver goods to several addresses. At end of day, each truck must end up back at depot so that it is ready to be loaded for next day. To reduce costs, company wants to select an order of delivery stops that yields lowest overall distance traveled by each truck. This problem is well-known "traveling-salesperson problem", & it is NP-complete [To be precise, only decision problems – those with a "yes/no" answer – can be NP-complete. Decision version of traveling salesperson problem asks whether there exists an order of stops whose distance totals at most a given amount.]. It has no known efficient algorithm. Under certain assumptions, however, we know of efficient algorithms that compute overall distances close to smallest possible. Chap. 35 discusses such "approximation algorithms".
– 1 ví dụ cụ thể, hãy xem xét 1 công ty giao hàng có 1 kho trung tâm. Mỗi ngày, công ty này chất hàng lên xe tải giao hàng tại kho & cử chúng đi giao hàng đến nhiều địa chỉ khác nhau. Vào cuối ngày, mỗi xe tải phải quay lại kho để sẵn sàng chất hàng cho ngày hôm sau. Để giảm chi phí, công ty muốn chọn thứ tự các điểm dừng giao hàng sao cho tổng quãng đường di chuyển của mỗi xe tải là nhỏ nhất. Bài toán này được biết đến là "bài toán nhân viên bán hàng lưu động", & là NP-complete [Nói 1 cách chính xác, chỉ những bài toán quyết định – những bài toán có câu trả lời "có/không" – mới có thể là NP-complete. Phiên bản quyết định của bài toán nhân viên bán hàng lưu động hỏi liệu có tồn tại thứ tự các điểm dừng có tổng quãng đường không vượt quá 1 số tiền nhất định hay không.]. Không có thuật toán hiệu quả nào được biết đến. Tuy nhiên, theo 1 số giả định nhất định, chúng ta biết về các thuật toán hiệu quả có thể tính toán tổng quãng đường gần với giá trị nhỏ nhất có thể. Chương 35 thảo luận về các "thuật toán xấp xỉ" như vậy.
Alternative computing models. For many years, we could count on processor clock speeds increasing at a steady rate. Physical limitations present a fundamental roadblock to ever-increasing clock speeds, chips run risk of melting once their clock speeds become high enough. In order to perform more computations per sec, therefore, chips are being designed to contain not just 1 but several processing "cores". We can liken these multicore computers to several sequential computers on a single chip. I.e., they are a type of "parallel computer". In order to elicit best performance from multicore computers, we need to design algorithms with parallelism in mind. Chap. 26 present a model for "task-parallel" algorithms, which take advantage of multiple processing cores. This model has advantages from both theoretical & practical standpoints, & many modern parallel-programming platforms embrace sth similar to this model of parallelism.
– **Các mô hình điện toán thay thế.** Trong nhiều năm, chúng ta có thể tin tưởng vào tốc độ xung nhịp của bộ xử lý tăng với tốc độ ổn định. Các hạn chế về mặt vật lý là rào cản cơ bản đối với tốc độ xung nhịp ngày càng tăng, chip có nguy cơ bị tan chảy khi tốc độ xung nhịp của chúng trở nên đủ cao. Do đó, để thực hiện nhiều phép tính hơn mỗi giây, chip đang được thiết kế để chứa không chỉ 1 mà là nhiều "lõi" xử lý. Chúng ta có thể ví những máy tính đa lõi này như 1 số máy tính tuần tự trên 1 chip duy nhất. Tức là chúng là 1 loại "máy tính song song". Để tạo ra hiệu suất tốt nhất từ máy tính đa lõi, chúng ta cần thiết kế các thuật toán có tính song song. Chương 26 trình bày 1 mô hình cho các thuật toán "song song tác vụ", tận dụng nhiều lõi xử lý. Mô hình này có những ưu điểm từ cả quan điểm lý thuyết & thực tế, & nhiều nền tảng lập trình song song hiện đại áp dụng thứ gì đó tương tự như mô hình song song này.
Most of examples in this book assume that all of input data are available when an algorithm begins running. Much of work in algorithm design makes same assumption. For many important real-world examples, however, input actually arrives over time, & algorithm must decide how to proceed without knowing what data will arrive in future. In a data center, jobs are constantly arriving & departing, & a scheduling algorithm must decide when & where to run a job, without knowing what jobs will be arriving in future. Traffic must be routed in internet based on current state, without knowing about where traffic will arrive in future. Hospital emergency rooms make triage decisions about which patients to treat 1st without knowing when other patients will be arriving in future & what treatments they will need. Algorithms that receive their input over time, rather than having al input present at start, are online algorithms, which Chap. 27 examines.
– Hầu hết các ví dụ trong cuốn sách này đều cho rằng tất cả dữ liệu đầu vào đều có sẵn khi thuật toán bắt đầu chạy. Phần lớn công việc thiết kế thuật toán đều đưa ra cùng 1 giả định. Tuy nhiên, đối với nhiều ví dụ thực tế quan trọng, đầu vào thực sự đến theo thời gian, & thuật toán phải quyết định cách tiến hành mà không biết dữ liệu nào sẽ đến trong tương lai. Trong 1 trung tâm dữ liệu, các công việc liên tục đến & khởi hành, & thuật toán lập lịch phải quyết định khi nào & chạy công việc ở đâu mà không biết công việc nào sẽ đến trong tương lai. Lưu lượng truy cập phải được định tuyến trên internet dựa trên trạng thái hiện tại mà không biết lưu lượng truy cập sẽ đến đâu trong tương lai. Các phòng cấp cứu của bệnh viện đưa ra quyết định phân loại về việc điều trị bệnh nhân nào trước mà không biết khi nào những bệnh nhân khác sẽ đến trong tương lai & họ sẽ cần phương pháp điều trị nào. Các thuật toán nhận đầu vào theo thời gian, thay vì có tất cả đầu vào khi bắt đầu, là các thuật toán trực tuyến, mà Chương 27 sẽ xem xét.

Problem 1 ([Cor+22], 1.1-1, p. 36)**.** *Describe your own real-world example that requires sorting. Describe one that requires finding shortest distance between 2 points.*
– Mô tả ví dụ thực tế của riêng bạn yêu cầu sắp xếp. Mô tả ví dụ yêu cầu tìm khoảng cách ngắn nhất giữa 2 điểm.

Problem 2 ([Cor+22], 1.1-2, p. 36)**.** *Other than speed, what other measures of efficiency might you need to consider in a real-world setting?*
– Ngoài tốc độ, bạn có thể cần cân nhắc những biện pháp hiệu quả nào khác trong bối cảnh thực tế?

Problem 3 ([Cor+22], 1.1-3, p. 36)**.** *Select a data structure that you have seen & discuss its strength & limitations.*
– Chọn 1 cấu trúc dữ liệu mà bạn đã thấy & thảo luận về điểm mạnh & hạn chế của nó.

Problem 4 ([Cor+22], 1.1-4, p. 36)**.** *How are the shortest-path & traveling-salesperson problems given above similar? How are they different?*
– Các bài toán về đường đi ngắn nhất & nhân viên bán hàng du lịch nêu trên có điểm gì giống và khác nhau? Chúng khác nhau như thế nào?

Problem 5 ([Cor+22], 1.1-5, p. 36)**.** *Suggest a real-world problem in which only the best solution will do. Then come up with one in which "approximately" best solution is good enough.*
– Đề xuất 1 vấn đề thực tế mà chỉ có giải pháp tốt nhất mới có thể giải quyết. Sau đó đưa ra 1 giải pháp mà giải pháp tốt nhất "xấp xỉ" là đủ tốt.

Problem 6 ([Cor+22], 1.1-6, p. 36)**.** *Describe a real-world problem in which sometimes entire input is available before you need to solve problem, but other times input is not entirely available in advance & arrives over time.*
– Mô tả 1 vấn đề thực tế trong đó đôi khi toàn bộ dữ liệu đầu vào có sẵn trước khi bạn cần giải quyết vấn đề, nhưng đôi khi dữ liệu đầu vào không có sẵn hoàn toàn trước & đến theo thời gian.

∗ 1.2. Algorithms as a technology. If computers were infinitely fast & computer memory were free, would you have any reason to study algorithms? Answer is yes, if for no other reason than that you would still like to be certain that your solution method terminates & does so with correct answer.
– Nếu máy tính vô cùng nhanh & bộ nhớ máy tính là miễn phí, bạn có lý do gì để nghiên cứu thuật toán không? Câu trả lời là có, nếu không vì lý do nào khác thì bạn vẫn muốn chắc chắn rằng phương pháp giải của bạn kết thúc & với câu trả lời đúng.
If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within bounds of good software engineering practice (e.g., your implementation should be well designed & documented), but you would most often use whichever method was easiest to implement.
– Nếu máy tính vô cùng nhanh, bất kỳ phương pháp đúng nào để giải quyết vấn đề đều có thể thực hiện được. Có lẽ bạn muốn việc triển khai của mình nằm trong giới hạn của thông lệ kỹ thuật phần mềm tốt (ví dụ: việc triển khai của bạn phải được thiết kế tốt & có tài liệu), nhưng bạn thường sẽ sử dụng bất kỳ phương pháp nào dễ triển khai nhất.
Of course, computers may be fast, but they are not infinitely fast. Computing time is therefore a bounded resource, which makes it precious. Although saying goes, "Time is money," time is even more valuable than money: you can get back money after you spend it, but once time is spent, you can never get it back. Memory may be inexpensive, but it is neither infinite nor free. You should choose algorithms that use resources of time & space efficiently.
– Tất nhiên, máy tính có thể nhanh, nhưng chúng không vô hạn. Do đó, thời gian tính toán là 1 nguồn tài nguyên hữu hạn, khiến nó trở nên quý giá. Mặc dù có câu nói, "Thời gian là tiền bạc", thời gian thậm chí còn có giá trị hơn tiền bạc: bạn có thể lấy lại tiền sau khi đã chi tiêu, nhưng 1 khi thời gian đã trôi qua, bạn sẽ không bao giờ có thể lấy lại được. Bộ nhớ có thể không tốn kém, nhưng nó không vô hạn cũng không miễn phí. Bạn nên chọn các thuật toán sử dụng hiệu quả các nguồn tài nguyên thời gian & không gian.
Efficiency. Different algorithms devised to solve same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware & software.
– **Hiệu quả.** Các thuật toán khác nhau được thiết kế để giải quyết cùng 1 vấn đề thường có hiệu quả khác nhau đáng kể. Những khác biệt này có thể quan trọng hơn nhiều so với những khác biệt do phần cứng & phần mềm.
E.g., Chap. 2 introduces 2 algorithms for sorting. 1st, known as insertion sort, takes time roughly equal to $c_1 n^2$ to sort n times, where c_1 is a constant that does not depend on n. I.e., it takes time roughly proportional to n^2. 2nd, merge sort, takes time roughly equal to $c_2 n \log_2 n$ & c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. Constant factors can have far less of an impact on running time than dependence on input size n. Write insertion sort's running time as $c_1 nn$ & merge sort's running time as $c_2 n \log_2 n$. Then see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\log_2 n$, which is much smaller. E.g., when $n = 1000$, $\log_2 n \approx 10$, & when $n = 10^6$, $\log_2 n \approx 20$. Although insertion sort usually runs faster than merge sort for small input sizes, once input size n becomes large enough, merge sort's advantage of $\log_2 n$ vs. n more than compensates for difference in constant factors. No matter how much smaller c_1 is than c_2, there is always a crossover point beyond which merge sort is faster.
– Ví dụ, Chương 2 giới thiệu 2 thuật toán để sắp xếp. Thuật toán đầu tiên, được gọi là sắp xếp chèn, mất thời gian gần bằng $c_1 n^2$ để sắp xếp n lần, trong đó c_1 là hằng số không phụ thuộc vào n. Tức là, mất thời gian gần bằng tỷ lệ thuận với n^2. Thuật toán thứ hai, sắp xếp trộn, mất thời gian gần bằng $c_2 n \log_2 n$ & c_2 là 1 hằng số khác cũng không phụ thuộc vào n. Sắp xếp chèn thường có hệ số hằng số nhỏ hơn sắp xếp trộn, do đó $c_1 < c_2$. Hệ số hằng số có thể có tác động ít hơn nhiều đến thời gian chạy so với sự phụ thuộc vào kích thước đầu vào n. Viết thời gian chạy của sắp xếp chèn là $c_1 nn$ & thời gian chạy của sắp xếp trộn là $c_2 n \log_2 n$. Sau đó, hãy xem rằng trong khi sắp xếp chèn có hệ số n trong thời gian chạy của nó, sắp xếp trộn có hệ số $\log_2 n$, nhỏ hơn nhiều. Ví dụ, khi $n = 1000$, $\log_2 n \approx 10$, & khi $n = 10^6$, $\log_2 n \approx 20$. Mặc dù sắp xếp chèn thường chạy nhanh hơn sắp xếp trộn đối với kích thước đầu vào nhỏ, nhưng khi kích thước đầu vào n trở nên đủ lớn, lợi thế của sắp xếp trộn là $\log_2 n$ so với n bù đắp cho sự khác biệt trong các hệ số hằng số. Bất kể c_1 nhỏ hơn c_2 bao nhiêu, luôn có 1 điểm giao nhau mà sau đó sắp xếp trộn sẽ nhanh hơn.
p. 38+++

◦ 2. Getting Started. This chap will familiarize you with framework use throughout book to think about design & analysis of algorithms. It is self-contained, but it does include several references to material that will be introduced in Chaps. 3–4. (It also contains several summations, which Appendix A shows how to solve.)
– Chương này sẽ giúp bạn làm quen với việc sử dụng khuôn khổ trong suốt cuốn sách để suy nghĩ về thiết kế & phân tích thuật toán. Chương này là chương độc lập, nhưng có bao gồm 1 số tài liệu tham khảo sẽ được giới thiệu trong Chương 3-4.

(Chương này cũng bao gồm 1 số phép tính tổng, Phụ lục A sẽ chỉ ra cách giải quyết.)

Begin by examining insertion sort algorithm to solve sorting problem introduced in Chap. 1. Specify algorithms using a pseudocode that should be understandable to you if you have done computer programming. See why insertion sort correct sorts & analyze its running time. Analysis introduces a notation that describes how running time increases with number of items to be sorted. Following a discussion of insertion sort, use divide-&-conquer to develop a sorting algorithm called merge sort. End with an analysis of merge sort's running time.

– Bắt đầu bằng cách xem xét thuật toán sắp xếp chèn để giải quyết vấn đề sắp xếp được giới thiệu trong Chương 1. Chỉ định các thuật toán bằng mã giả mà bạn có thể hiểu được nếu bạn đã học lập trình máy tính. Xem lý do tại sao sắp xếp chèn sắp xếp đúng & phân tích thời gian chạy của nó. Phân tích giới thiệu 1 ký hiệu mô tả cách thời gian chạy tăng theo số lượng mục cần sắp xếp. Sau khi thảo luận về sắp xếp chèn, hãy sử dụng chia-&-chinh phục để phát triển 1 thuật toán sắp xếp được gọi là sắp xếp trộn. Kết thúc bằng phân tích thời gian chạy của sắp xếp trộn.

∗ 2.1. Insertion sort. 1st algorithm, insertion sort, solves sorting problem.

Problem 7 (Sorting – Sắp xếp)**.**

Input. *A sequence of $n \in \mathbb{N}^\star$ numbers $a_1, \ldots, a_n$.*

Output. *A permutation (reordering) $a'_1, \ldots, a'_n$ of input sequence s.t. $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.*

Numbers to be sorted are also known as *keys*. Although problem is conceptually about sorting a sequence, input comes in form of an array with n elements. When want to sort numbers, it's often because they are keys associated with other data, which we call *satellite data*. Together, a key & satellite data form a *record*. E.g., consider a spreadsheet containing student records with many associated pieces of data e.g. age, grade-point average, & number of courses taken. Any 1 of these quantities could be a key, but when spreadsheet sorts, it moves associated record (satellite data) with key. When describing a sorting algorithm, focus on keys, but important to remember: there usually is associated satellite data.

– Các số cần sắp xếp cũng được gọi là *keys*. Mặc dù về mặt khái niệm, vấn đề là về việc sắp xếp 1 chuỗi, nhưng dữ liệu đầu vào có dạng 1 mảng với n phần tử. Khi muốn sắp xếp các số, thường là do chúng là các khóa được liên kết với dữ liệu khác, mà chúng ta gọi là *dữ liệu vệ tinh*. Cùng nhau, 1 khóa & dữ liệu vệ tinh tạo thành 1 *bản ghi*. Ví dụ: hãy xem xét 1 bảng tính chứa hồ sơ sinh viên với nhiều phần dữ liệu liên quan, ví dụ như tuổi, điểm trung bình, & số khóa học đã học. Bất kỳ 1 trong số các số lượng này đều có thể là 1 khóa, nhưng khi bảng tính sắp xếp, nó sẽ di chuyển bản ghi liên quan (dữ liệu vệ tinh) cùng với khóa. Khi mô tả thuật toán sắp xếp, hãy tập trung vào các khóa, nhưng điều quan trọng cần nhớ: thường có dữ liệu vệ tinh liên quan.

Remark 1. *[Knu98] explains more rigorous with mathematical definitions & conditions.*

In this book, typically describe algorithms as procedures written in a pseudocode that is similar in many respects to C, C++, Java, Python, [If familiar with only Python, can think of arrays as similar to Python lists.], or JavaScript. If have been introduced to any of these languages, should have little trouble understanding algorithms "coded" in pseudocode. What separates pseudocode from real code: in pseudocode, employ whatever expressive method is most clear & concise to specify a given algorithm. Sometimes clearest method is English, so do not surprised if you come across an English phrase or sentence embedded within a sect that looks more like real code. Another difference between pseudocode & real code: pseudocode often ignores aspects of software engineering – e.g. data abstraction, modularity, & error handling – in order to convey essence of algorithm more concisely.

– Trong cuốn sách này, thông thường mô tả các thuật toán như các thủ tục được viết bằng mã giả tương tự về nhiều mặt với C, C++, Java, Python, [Nếu chỉ quen với Python, có thể coi mảng tương tự như danh sách Python.], hoặc JavaScript. Nếu đã được giới thiệu về bất kỳ ngôn ngữ nào trong số các ngôn ngữ này, sẽ không gặp khó khăn gì khi hiểu các thuật toán "được mã hóa" trong mã giả. Điểm khác biệt giữa mã giả và mã thực: trong mã giả, sử dụng bất kỳ phương pháp diễn đạt nào rõ ràng nhất & súc tích nhất để chỉ định 1 thuật toán nhất định. Đôi khi phương pháp rõ ràng nhất là tiếng Anh, vì vậy đừng ngạc nhiên nếu bạn bắt gặp 1 cụm từ hoặc câu tiếng Anh được nhúng trong 1 phần trông giống mã thực hơn. Một điểm khác biệt nữa giữa mã giả & mã thực: mã giả thường bỏ qua các khía cạnh của kỹ thuật phần mềm – ví dụ: trừu tượng hóa dữ liệu, tính mô-đun, & xử lý lỗi – để truyền đạt bản chất của thuật toán 1 cách súc tích hơn.

Start with insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works way you might sort a hand of playing cards. Start with an empty left hand & cards in a pile on table. Pick up 1st card in pile & hole it with your left hand. Then, with your right hand, remove 1 card at a time from pile, & insert it into correct position in your left hand. As Fig. 2.1: Sorting a hand of cards using insertion sort illustrates, find correct position for a card by comparing it with each of cards already in your left hand, starting at right & moving left. As soon as you see a card in your left hand whose value is $\leq$ card you're holding in your right hand, insert card that you're holding in your right hand just to right of this card in your left hand. If all cards in your left hand have values $>$ card in your right hand, then place this card as leftmost card in your left hand. At all times, cards held in your left hand are sorted, & these cards were originally top cards of pile on table.

– Bắt đầu với sắp xếp chèn, đây là 1 thuật toán hiệu quả để sắp xếp 1 số lượng nhỏ các phần tử. Sắp xếp chèn hoạt động theo cách bạn có thể sắp xếp 1 ván bài. Bắt đầu với 1 tay trái trống & các lá bài trong 1 chồng trên bàn. Nhặt lá bài đầu tiên trong chồng & đục lỗ bằng tay trái của bạn. Sau đó, bằng tay phải của bạn, lấy từng lá bài 1 khỏi chồng, & chèn nó vào đúng vị trí trong tay trái của bạn. Như Hình 2.1: Sắp xếp 1 ván bài bằng sắp xếp chèn minh họa, hãy tìm vị trí đúng cho 1 lá bài bằng cách so sánh nó với từng lá bài đã có trong tay trái của bạn, bắt đầu từ bên phải & di chuyển sang trái. Ngay khi bạn nhìn thấy 1 lá bài trong tay trái của bạn có giá trị là $\leq$ lá bài bạn đang cầm trong tay phải, hãy chèn lá bài mà bạn đang cầm trong tay phải của bạn ngay bên phải lá bài này trong tay trái của bạn. Nếu tất cả các lá bài trong tay trái của bạn có giá trị $>$ lá bài trong tay phải của bạn, thì hãy đặt lá bài này làm lá bài ngoài cùng bên trái

của bạn trong tay trái của bạn. Mọi lúc, các lá bài bạn cầm trên tay trái đều được phân loại & những lá bài này ban đầu là những lá bài ở trên cùng của chồng bài trên bàn.

Pseudocode for insertion sort is given as procedure INSERTION-SORT. It takes 2 parameters: an array A containing values to be sorted & number n of values of sort. Values occupy positions $A[1]$ through $A[n]$ of array, which we denote by $A[1:n]$. When INSERTION-SORT procedure is finished, array $A[1:n]$ contains original values, but in sorted order.

– Mã giả cho sắp xếp chèn được đưa ra dưới dạng thủ tục INSERTION-SORT. Nó lấy 2 tham số: 1 mảng A chứa các giá trị cần sắp xếp & số n giá trị cần sắp xếp. Các giá trị chiếm các vị trí từ $A[1]$ đến $A[n]$ của mảng, mà chúng ta ký hiệu là $A[1:n]$. Khi thủ tục INSERTION-SORT hoàn tất, mảng $A[1:n]$ chứa các giá trị gốc, nhưng theo thứ tự đã sắp xếp.

INSERTION-SORT(A, n)

```
1  for i = 2 to n
2      key = A[i]
3      // insert A[i] into sorted subarray A[1:i - 1]
4      j = i - 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j - 1
8      A[j + 1] = key
```

Loop invariants & correctness of insertion sort. p. 47+++

∗ 2.2. Analyzing algorithms. Analyzing an algorithm has come to mean predicting resources that algorithm requires. You might consider resources e.g. memory, communication bandwidth, or energy consumption. Most often, however, want to measure computational time. If analyze several candidate algorithms for a problem, can identify most efficient one. There might be more than just 1 viable candidate, but you can often rule out several inferior algorithms in process.

– Phân tích 1 thuật toán có nghĩa là dự đoán các tài nguyên mà thuật toán yêu cầu. Bạn có thể xem xét các tài nguyên như bộ nhớ, băng thông truyền thông hoặc mức tiêu thụ năng lượng. Tuy nhiên, thường thì muốn đo thời gian tính toán. Nếu phân tích 1 số thuật toán ứng viên cho 1 vấn đề, có thể xác định được thuật toán hiệu quả nhất. Có thể có nhiều hơn 1 ứng viên khả thi, nhưng bạn thường có thể loại trừ 1 số thuật toán kém hơn đang trong quá trình xử lý.

Before you can analyze an algorithm, need a model of technology that it runs on, including resources of that technology & a way to express their costs. Most of this book assumes a generic 1-processor, random-access machine (RAM) model of computation as implementation technology, with understanding that algorithms are implemented as computer programs. In RAM model, instructions execute 1 after another, with no concurrent operations. RAM model assumes: each instruction takes same amount of time as any other instruction & that each data access – using value of a variable or storing into a variable – takes same amount of time as any other data access. I.e., in RAM model each instruction or data access takes a constant amount of time – even indexing into an array [Assume: each element of a given array occupies same number of bytes & elements of a given array are stored in contiguous memory locations. E.g., if array $A[1:n]$ starts at memory address 1000 & each element occupies 4 bytes, then element $A[i]$ is at address $1000 + 4(i-1)$. In general, computing address in memory of a particular array element requires at most 1 subtraction (no subtraction for a 0-origin array), 1 multiplication (often implemented as a shift operation if element size is an exact power of 2), & 1 addition. Furthermore, for code that iterates through elements of an array in order, an optimizing compiler can generate address of each element using just 1 addition, by adding element size to address of preceding element.].

– Trước khi bạn có thể phân tích 1 thuật toán, cần có 1 mô hình công nghệ mà thuật toán chạy trên đó, bao gồm các tài nguyên của công nghệ đó & 1 cách để thể hiện chi phí của chúng. Hầu hết cuốn sách này giả định 1 mô hình máy tính truy cập ngẫu nhiên (RAM) 1 bộ xử lý chung làm công nghệ triển khai, với sự hiểu biết rằng các thuật toán được triển khai dưới dạng các chương trình máy tính. Trong mô hình RAM, các lệnh thực thi 1 lệnh sau lệnh khác, không có hoạt động đồng thời. Mô hình RAM giả định: mỗi lệnh mất cùng 1 khoảng thời gian như bất kỳ lệnh nào khác & rằng mỗi lần truy cập dữ liệu – sử dụng giá trị của 1 biến hoặc lưu trữ vào 1 biến – mất cùng 1 khoảng thời gian như bất kỳ lần truy cập dữ liệu nào khác. Tức là, trong mô hình RAM, mỗi lệnh hoặc truy cập dữ liệu mất 1 khoảng thời gian không đổi – ngay cả khi lập chỉ mục vào 1 mảng [Giả sử: mỗi phần tử của 1 mảng nhất định chiếm cùng 1 số byte & các phần tử của 1 mảng nhất định được lưu trữ ở các vị trí bộ nhớ liền kề. Ví dụ, nếu mảng $A[1:n]$ bắt đầu tại địa chỉ bộ nhớ 1000 & mỗi phần tử chiếm 4 byte, thì phần tử $A[i]$ nằm tại địa chỉ $1000 + 4(i-1)$. Nhìn chung, việc tính toán địa chỉ trong bộ nhớ của 1 phần tử mảng cụ thể chỉ cần tối đa 1 phép trừ (không cần phép trừ đối với mảng có gốc 0), 1 phép nhân (thường được triển khai dưới dạng phép dịch chuyển nếu kích thước phần tử là lũy thừa chính xác của 2), & 1 phép cộng. Hơn nữa, đối với mã lặp qua các phần tử của 1 mảng theo thứ tự, trình biên dịch tối ưu hóa có thể tạo địa chỉ của mỗi phần tử chỉ bằng 1 phép cộng, bằng cách thêm kích thước phần tử vào địa chỉ của phần tử trước đó.].

Strictly speaking, should precisely define instructions of RAM model & their costs. To do so, however, would be tedious & yield little insight into algorithm design & analysis. Yet must be careful not to abuse RAM model. E.g., what if a RAM had an instruction that sorts? Then you could sort in just 1 step. Such a RAM would be unrealistic, since such instructions do not appear in real computers. Our guide, therefore, is how real computers are designed. RAM model contains instructions commonly found in real computers: arithmetic (e.g. add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), & control (conditional & unconditional branch, subroutine call & return).

– Nói 1 cách nghiêm ngặt, cần định nghĩa chính xác các lệnh của mô hình RAM & chi phí của chúng. Tuy nhiên, để làm như vậy sẽ rất tẻ nhạt & mang lại ít hiểu biết về thiết kế thuật toán & phân tích. Tuy nhiên, phải cẩn thận không lạm dụng mô hình RAM. Ví dụ, nếu RAM có lệnh sắp xếp thì sao? Khi đó, bạn có thể sắp xếp chỉ trong 1 bước. Một RAM như vậy sẽ không thực tế, vì các lệnh như vậy không xuất hiện trong máy tính thực. Do đó, hướng dẫn của chúng tôi là

cách thiết kế máy tính thực. Mô hình RAM chứa các lệnh thường thấy trong máy tính thực: số học (ví dụ: cộng, trừ, nhân, chia, phần dư, làm tròn, làm tròn trần), di chuyển dữ liệu (tải, lưu trữ, sao chép), & điều khiển (nhánh có điều kiện & không điều kiện, gọi chương trình con & trả về).
Data types in RAM model are integer, floating point (for storing real-number approximations), & character. Real computers do not usually have a separate data type for boolean values TRUE & FALSE. Instead, they often test whether an integer value is 0 (FALSE) or nonzero (TRUE), as in C. Although we typically do not concern with precision for floating-point values in this book (many numbers cannot be represented exactly in floating point), precision is crucial for most applications. Also assume: each word of data has a limit on number of bits. E.g., when working with inputs of size n, typically assume: integers are represented by $c \log_2 n$ bits for some constants $c \geq 1$. Require $c \geq 1$ so that each word can hold value of n, enabling us to index individual input elements, & restrict c to be a constant so that word size does not grow arbitrarily. (If word size could grow arbitrarily, could store huge amounts of data in 1 word & operate on it all in constant time – an unrealistic scenario.)
– Các kiểu dữ liệu trong mô hình RAM là số nguyên, số thực (để lưu trữ các giá trị xấp xỉ số thực), & ký tự. Máy tính thực tế thường không có kiểu dữ liệu riêng cho các giá trị boolean TRUE & FALSE. Thay vào đó, chúng thường kiểm tra xem 1 giá trị số nguyên là 0 (FALSE) hay khác không (TRUE), như trong C. Mặc dù chúng ta thường không quan tâm đến độ chính xác đối với các giá trị dấu phẩy động trong cuốn sách này (nhiều số không thể được biểu diễn chính xác trong dấu phẩy động), nhưng độ chính xác là rất quan trọng đối với hầu hết các ứng dụng. Cũng giả sử: mỗi từ dữ liệu có giới hạn về số bit. Ví dụ: khi làm việc với các đầu vào có kích thước n, thường giả sử: các số nguyên được biểu diễn bằng $c \log_2 n$ bit cho 1 số hằng số $c \geq 1$. Yêu cầu $c \geq 1$ để mỗi từ có thể chứa giá trị n, cho phép chúng ta lập chỉ mục các phần tử đầu vào riêng lẻ, & hạn chế c thành 1 hằng số để kích thước từ không tăng tùy ý. (Nếu kích thước từ có thể tăng tùy ý, có thể lưu trữ lượng dữ liệu khổng lồ trong 1 từ & xử lý tất cả trong thời gian không đổi – 1 kịch bản không thực tế.)
Real computers contain instructions not listed above, & such instructions represent a gray area in RAM model. E.g., is exponentiation a constant-time instruction? In general case, no: to compute x^n when x & n are general integers takes time logarithmic in n, & must worry about whether result fits into a computer word. If n is an exact power of 2, however, exponentiation can usually be viewed as a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts bits of an integer by n positions to left. In most computers, shifting bits of an integer by 1 position to left is equivalent to multiplying by 2, so that shifting bits by n positions to left is equivalent to multiplying by 2^n. Therefore, such computers can compute 2^n in 1 constant-time instruction by shifting integer 1 by n positions to left, as long as n is no more than number of bits in a computer word. Try to avoid such gray areas in RAM model & treat computing 2^n & multiplying by 2^n as constant-time operations when result is small enough to fit in a computer word.
– Máy tính thực tế chứa các lệnh không được liệt kê ở trên, & các lệnh như vậy biểu thị 1 vùng xám trong mô hình RAM. Ví dụ, lũy thừa có phải là lệnh thời gian hằng số không? Trong trường hợp chung, không: để tính x^n khi x & n là các số nguyên tổng quát mất thời gian logarit theo n, & phải quan tâm đến việc kết quả có phù hợp với 1 từ máy tính hay không. Tuy nhiên, nếu n là lũy thừa chính xác của 2, thì lũy thừa thường có thể được xem là 1 phép toán thời gian hằng số. Nhiều máy tính có lệnh "dịch chuyển sang trái", trong thời gian hằng số, dịch chuyển các bit của 1 số nguyên n vị trí sang trái. Trong hầu hết các máy tính, dịch chuyển các bit của 1 số nguyên 1 vị trí sang trái tương đương với nhân với 2, do đó, dịch chuyển các bit n vị trí sang trái tương đương với nhân với 2^n. Do đó, các máy tính như vậy có thể tính 2^n trong 1 lệnh thời gian hằng số bằng cách dịch chuyển số nguyên 1 n vị trí sang trái, miễn là n không quá số bit trong 1 từ máy tính. Cố gắng tránh những vùng xám như vậy trong mô hình RAM & coi việc tính toán 2^n & nhân với 2^n là các phép toán thời gian hằng số khi kết quả đủ nhỏ để có thể vừa với 1 từ trong máy tính.
RAM model does not account for memory hierarchy that is common in contemporary computers. It models neither caches nor virtual memory. Several other computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. Sect. 11.5 & a handful of problems in this book examine memory-hierarchy effects, but for most part, analyses in this book do not consider them. Models that include memory hierarchy are quite a bit more complex than RAM model, & so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.
– Mô hình RAM không tính đến hệ thống phân cấp bộ nhớ phổ biến trong các máy tính hiện đại. Nó không mô hình hóa bộ nhớ đệm cũng như bộ nhớ ảo. Một số mô hình tính toán khác cố gắng tính đến các hiệu ứng phân cấp bộ nhớ, đôi khi có ý nghĩa quan trọng trong các chương trình thực trên máy thực. Phần 11.5 & 1 số ít các vấn đề trong cuốn sách này xem xét các hiệu ứng phân cấp bộ nhớ, nhưng phần lớn các phân tích trong cuốn sách này không xem xét chúng. Các mô hình bao gồm hệ thống phân cấp bộ nhớ phức tạp hơn nhiều so với mô hình RAM, & vì vậy chúng có thể khó làm việc. Hơn nữa, các phân tích mô hình RAM thường là những công cụ dự đoán tuyệt vời về hiệu suất trên các máy thực tế.
Although often straightforward to analyze an algorithm in RAM model, sometimes it can be quite a challenge. Might need to employ mathematical tools e.g. combinatorics, probability theory, algebraic dexterity, & ability to identify most significant terms in a formula. Because an algorithm might behave differently for each possible input, need a means for summarizing that behavior in simple, easily understood formulas.
– Mặc dù thường dễ dàng phân tích 1 thuật toán trong mô hình RAM, đôi khi có thể khá khó khăn. Có thể cần sử dụng các công cụ toán học như tổ hợp, lý thuyết xác suất, sự khéo léo về đại số, & khả năng xác định các thuật ngữ quan trọng nhất trong 1 công thức. Vì 1 thuật toán có thể hoạt động khác nhau đối với mỗi đầu vào khả thi, cần có phương tiện để tóm tắt hành vi đó trong các công thức đơn giản, dễ hiểu.
Analysis of insertion sort. How long does INSERTION-SORT procedure take? 1 way to tell would be for you to run it on your computer & time how long it takes to run. Of course, 1st have to implement it in a real programming language, since you cannot run our pseudocode directly. What would such a timing test tell you? Find out how long insertion sort

takes to run on your particular computer, on that particular input, under particular implementation that you created, with particular compiler or interpreter that you ran, with particular libraries that you linked in, & with particular background tasks that were running on your computer concurrently with your timing test (e.g. checking for incoming information over a network). If run insertion sort again on your computer with same input, might even get a different timing result. From running just 1 implementation of insertion sort on just 1 computer & on just 1 input, what would you be able to determine about insertion sort's running time if you were to give it a different input, if you were to run it on a different computer, or if you were to implement it in a different programming language? Not much. Need a way to predict, given a new input, how long insertion sort will take.

– **Phân tích thuật toán sắp xếp chèn.** Thủ tục INSERTION-SORT mất bao lâu? Một cách để biết là bạn chạy nó trên máy tính của mình & tính thời gian chạy. Tất nhiên, trước tiên phải triển khai nó trong 1 ngôn ngữ lập trình thực, vì bạn không thể chạy trực tiếp mã giả của chúng tôi. Một bài kiểm tra thời gian như vậy sẽ cho bạn biết điều gì? Tìm hiểu thời gian chạy thuật toán sắp xếp chèn trên máy tính cụ thể của bạn, trên đầu vào cụ thể đó, dưới triển khai cụ thể mà bạn đã tạo, với trình biên dịch hoặc trình thông dịch cụ thể mà bạn đã chạy, với các thư viện cụ thể mà bạn đã liên kết, & với các tác vụ nền cụ thể đang chạy trên máy tính của bạn đồng thời với bài kiểm tra thời gian của bạn (ví dụ: kiểm tra thông tin đến qua mạng). Nếu chạy lại thuật toán sắp xếp chèn trên máy tính của bạn với cùng 1 đầu vào, thậm chí có thể nhận được kết quả thời gian khác. Từ việc chỉ chạy 1 triển khai sắp xếp chèn trên chỉ 1 máy tính & trên chỉ 1 đầu vào, bạn sẽ có thể xác định được điều gì về thời gian chạy của sắp xếp chèn nếu bạn cung cấp cho nó 1 đầu vào khác, nếu bạn chạy nó trên 1 máy tính khác hoặc nếu bạn triển khai nó trong 1 ngôn ngữ lập trình khác? Không nhiều. Cần 1 cách để dự đoán, với 1 đầu vào mới, sắp xếp chèn sẽ mất bao lâu.

Instead of timing a run, or even several runs, of insertion sort, can determine how long it takes by analyzing algorithm itslef. Examine how many times it executes each line of pseudocode & how long each line of pseudocode takes to run. 1st come up with a precise but complicated formula for running time. Then, distill important part of formula using a convenient notation that can help us compare running times of different algorithms for same problem.

– Thay vì tính thời gian chạy 1 lần, hoặc thậm chí nhiều lần chạy, của sắp xếp chèn, có thể xác định thời gian chạy bằng cách phân tích chính thuật toán. Kiểm tra xem nó thực thi mỗi dòng mã giả bao nhiêu lần & thời gian chạy mỗi dòng mã giả. Đầu tiên, hãy đưa ra 1 công thức chính xác nhưng phức tạp để tính thời gian chạy. Sau đó, chắt lọc phần quan trọng của công thức bằng cách sử dụng ký hiệu thuận tiện có thể giúp chúng ta so sánh thời gian chạy của các thuật toán khác nhau cho cùng 1 vấn đề.

How do we analyze insertion sort? 1st, acknowledge: running time depends on input. Shouldn't be terribly surprised that sorting a thousand numbers takes longer than sorting 3 numbers. Moreover, insertion sort can take different amounts of time to sort 2 input arrays of same size, depending on how nearly sorted they already are. Even though running time can depend on many features of input, focus on the one that has been shown to have greatest effect, namely size of input, & describe running time of a program as a function of size of its input. To do so, need to define terms "running time" & "input size" more carefully. Also need to be clear about whether we are discussing running time for an input that elicits worst-case behavior, best-case behavior, or some other case.

– Chúng ta phân tích thuật toán sắp xếp chèn như thế nào? Đầu tiên, hãy thừa nhận: thời gian chạy phụ thuộc vào đầu vào. Không nên quá ngạc nhiên khi sắp xếp 1 nghìn số mất nhiều thời gian hơn sắp xếp 3 số. Hơn nữa, thuật toán sắp xếp chèn có thể mất nhiều thời gian khác nhau để sắp xếp 2 mảng đầu vào có cùng kích thước, tùy thuộc vào mức độ sắp xếp gần đúng của chúng. Mặc dù thời gian chạy có thể phụ thuộc vào nhiều tính năng của đầu vào, hãy tập trung vào tính năng đã được chứng minh là có hiệu ứng lớn nhất, cụ thể là kích thước đầu vào, & mô tả thời gian chạy của chương trình như 1 hàm của kích thước đầu vào của nó. Để làm như vậy, cần phải định nghĩa các thuật ngữ "thời gian chạy" & "kích thước đầu vào" cẩn thận hơn. Ngoài ra, cần phải làm rõ liệu chúng ta đang thảo luận về thời gian chạy cho 1 đầu vào tạo ra hành vi trường hợp xấu nhất, hành vi trường hợp tốt nhất hay 1 số trường hợp khác.

Best notion for input size depends on problem being studied. For many problems, e.g. sorting or computing discrete Fourier transforms, most natural measure is *number of items in input* – e.g., number n of items being sorted. For many other problems, e.g. multiplying 2 integers, best measure of input size is *total number of bits* needed to represent input in ordinary binary notation. Sometimes it is more appropriate to describe size of input with more than just 1 number. E.g., if input to an algorithm is a graph, we usually characterize input size by both number of vertices & number of edges in graph. Indicate which input size measure is being used with each problem we study.

– Khái niệm tốt nhất cho kích thước đầu vào phụ thuộc vào vấn đề đang được nghiên cứu. Đối với nhiều vấn đề, ví dụ như sắp xếp hoặc tính toán biến đổi Fourier rời rạc, phép đo tự nhiên nhất là *số mục trong đầu vào* – ví dụ, số n mục đang được sắp xếp. Đối với nhiều vấn đề khác, ví dụ như nhân 2 số nguyên, phép đo tốt nhất cho kích thước đầu vào là *tổng số bit* cần thiết để biểu diễn đầu vào theo ký hiệu nhị phân thông thường. Đôi khi, việc mô tả kích thước đầu vào bằng nhiều hơn 1 số là phù hợp hơn. Ví dụ, nếu đầu vào cho 1 thuật toán là 1 đồ thị, chúng ta thường mô tả kích thước đầu vào bằng cả số đỉnh & số cạnh trong đồ thị. Chỉ ra phép đo kích thước đầu vào nào đang được sử dụng với từng vấn đề chúng ta nghiên cứu.

The *running time* of an algorithm on a particular input is number of instructions & data accesses executed. How we account for these costs should be independent of any particular computer, but within framework of RAM model. For moment, adopt following view. A constant amount of time is required to execute each line of our pseudocode. 1 line might take more or less time than another line, but assume: each execution of kth line takes c_k time, where c_k is a constant. This viewpoint is in keeping with RAM model, & it also reflects how pseudocode would be implemented on most actual computers. [There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. E.g., in RADIX-SORT procedure on p. 213, 1 line reads "use a stable sort to sort array A on digit i," which takes more than a constant amount of time. Also, although

a statement that calls a subroutine takes only constant time, subroutine itself, once invoked, may take more. I.e., we separate process of calling subroutines – passing parameters to it, etc. – from process of executing subroutine.]
– *thời gian chạy* của 1 thuật toán trên 1 đầu vào cụ thể là số lệnh & truy cập dữ liệu được thực thi. Cách chúng ta tính toán những chi phí này phải độc lập với bất kỳ máy tính cụ thể nào, nhưng phải nằm trong khuôn khổ của mô hình RAM. Hiện tại, hãy áp dụng quan điểm sau. Cần 1 lượng thời gian không đổi để thực thi từng dòng mã giả của chúng ta. 1 dòng có thể mất nhiều thời gian hơn hoặc ít thời gian hơn dòng khác, nhưng hãy giả sử: mỗi lần thực thi dòng thứ k mất c_k thời gian, trong đó c_k là 1 hằng số. Quan điểm này phù hợp với mô hình RAM, & nó cũng phản ánh cách mã giả sẽ được triển khai trên hầu hết các máy tính thực tế. [Có 1 số điểm tinh tế ở đây. Các bước tính toán mà chúng ta chỉ định bằng tiếng Anh thường là các biến thể của 1 quy trình đòi hỏi nhiều hơn 1 lượng thời gian không đổi. Ví dụ: trong quy trình RADIX-SORT ở trang 213, 1 dòng có nội dung "sử dụng sắp xếp ổn định để sắp xếp mảng A theo chữ số i," mất nhiều hơn 1 lượng thời gian không đổi. Ngoài ra, mặc dù 1 câu lệnh gọi 1 chương trình con chỉ mất thời gian không đổi, nhưng bản thân chương trình con, 1 khi được gọi, có thể mất nhiều thời gian hơn. Tức là, chúng ta tách quá trình gọi chương trình con – truyền tham số cho nó, v.v. – khỏi quá trình thực thi chương trình con.]
Analyze INSERTION-SORT procedure. As promised, start by devising a precise formula that uses input size & all statement costs c_k. This formula turns out to be messy, however. Then switch to a simpler notation that is more concise & easier to use. This simpler notation makes clear how to compare running times of algorithms, especially as size of input increases.
– Phân tích quy trình INSERTION-SORT. Như đã hứa, hãy bắt đầu bằng cách thiết kế 1 công thức chính xác sử dụng kích thước đầu vào & tất cả các câu lệnh có chi phí c_k. Tuy nhiên, công thức này hóa ra lại lộn xộn. Sau đó chuyển sang ký hiệu đơn giản hơn, súc tích hơn & dễ sử dụng hơn. Ký hiệu đơn giản hơn này làm rõ cách so sánh thời gian chạy của các thuật toán, đặc biệt là khi kích thước đầu vào tăng lên.
To analyze INSERTION-SORT procedure, view it with time cost of each statement & number of times each statement is executed. for each $i = 2, 3, \ldots, n$, let t_i denote number of times `while` loop test in line 5 is executed for that value of i. When a `for` or `while` loop exits in usual way – because test in loop header comes up FALSE – test is executed 1 time more than loop body. Because comments are not executable statements, assume they take no time.
– Để phân tích thủ tục INSERTION-SORT, hãy xem thủ tục này với chi phí thời gian của mỗi câu lệnh & số lần mỗi câu lệnh được thực thi. Đối với mỗi $i = 2, 3, \ldots, n$, hãy để t_i biểu thị số lần vòng lặp `while` được thực thi cho giá trị i đó. Khi vòng lặp `for` hoặc `while` thoát theo cách thông thường – vì vòng lặp kiểm tra trong tiêu đề trả về FALSE – thì vòng lặp kiểm tra được thực thi nhiều hơn 1 lần so với thân vòng lặp. Vì chú thích không phải là câu lệnh thực thi, hãy cho rằng chúng không mất thời gian.
Running time of algorithm is sum of running times for each statement executed. A statement that takes c_k steps to execute & executes m times contributes $c_k m$ to total running time. [This characteristic does not necessarily hold for a resource e.g. memory. A statement that references m words of memory & is executed n times does not necessarily reference mn distinct words of memory.] Usually denote running time of an algorithm on an input size n by $T(n)$. To compute $T(n)$, running time of INSERTION-SORT on an input of n values, sum products of *cost & times* columns, obtaining
– Thời gian chạy của thuật toán là tổng thời gian chạy cho mỗi câu lệnh được thực thi. Một câu lệnh mất c_k bước để thực thi & thực thi m lần đóng góp $c_k m$ vào tổng thời gian chạy. [Đặc điểm này không nhất thiết phải đúng đối với 1 tài nguyên như bộ nhớ. Một câu lệnh tham chiếu m từ bộ nhớ & được thực thi n lần không nhất thiết phải tham chiếu mn từ bộ nhớ riêng biệt.] Thường biểu thị thời gian chạy của 1 thuật toán trên đầu vào có kích thước n bằng $T(n)$. Để tính $T(n)$, thời gian chạy của INSERTION-SORT trên đầu vào có n giá trị, hãy tính tổng tích của *cost & times* cột, thu được

```
1  INSERTION-SORT(A, n) // costtimes
2  for i = 2 to n // c1 n
3      key = A[i] // c2 n - 1
4      // insert A[i] into sorted subarray A[1:i - 1] // 0 n - 1
5      j = i - 1 // c4 n - 1
6      while j > 0 and A[j] > key // c5 \sum_{i=2}^n t_i
7          A[j + 1] = A[j] // c6 \sum_{i=2}^n (t_i - 1)
8          j = j - 1 // c7 \sum_{i=2}^n (t_i - 1)
9      A[j + 1] = key // c8 n - 1
```

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1).$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. E.g., in INSERTION-SORT, best case occurs when array is already sorted. In this case, each time that line 5 executes, value of *key* – value originally in $A[i]$ – is already $\geq$ all values in $A[1 : i-1]$, so that `while` loop of lines 5–7 always exits upon 1st test in line 5. Therefore, have $t_i = 1$ for $i = 2, 3, \ldots, n$, & best-case running time is given by

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Can express this running time as $an + b$ for constants a, b that depend on statement costs c_k (where $a = c_1 + c_2 + c_4 + c_5 + c_8, b = -(c_2 + c_4 + c_5 + c_8)$). Running time is thus a *linear function* of n.
– Ngay cả đối với các đầu vào có kích thước nhất định, thời gian chạy của thuật toán có thể phụ thuộc vào *đầu vào nào* có kích thước đó được cung cấp. E.g., trong INSERTION-SORT, trường hợp tốt nhất xảy ra khi mảng đã được sắp xếp. Trong trường hợp này, mỗi lần dòng 5 thực thi, giá trị của *key* – giá trị ban đầu trong $A[i]$ – đã là $\geq$ tất cả các giá trị

trong $A[1:i-1]$, do đó vòng lặp `while` của các dòng 5–7 luôn thoát khi kiểm tra lần đầu tiên ở dòng 5. Do đó, có $t_i = 1$ với $i = 2, 3, \ldots, n$, & thời gian chạy tốt nhất được đưa ra bởi

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Có thể biểu thị thời gian chạy này là $an + b$ đối với các hằng số a, b phụ thuộc vào chi phí câu lệnh c_k (trong đó $a = c_1 + c_2 + c_4 + c_5 + c_8, b = -(c_2 + c_4 + c_5 + c_8)$). Do đó, thời gian chạy là 1 *hàm tuyến tính* của n.

Worst case arises when array is in reverse sorted order – i.e., it starts out in decreasing order. Procedure must compare each element $A[i]$ with each element in entire sorted subarray $A[1:i-1]$, & so $t_i = i$ for $i = 2, \ldots, n$. (Procedure finds $A[j] > key$ every time in line 5, & `while` loop exits only when j reaches 0.) Note $\sum_{i=2}^{n} i = (\sum_{i=1}^{n} i) - 1 = \frac{n(n+1)}{2} - 1$ & $\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, find in worst case, running time of INSERTION-SORT is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \Big(c_1 + c_2 + c_4 +$$

Can express this worst-case running time as $an^2 + bn + c$ for constants a, b, c that again depend on statement costs c_k (now $a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}, b = c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8, c = -(c_2 + c_4 + c_5 + c_8)$). Running time is thus a *quadratic function* of n.

– Trường hợp xấu nhất xảy ra khi mảng được sắp xếp theo thứ tự ngược lại – tức là, nó bắt đầu theo thứ tự giảm dần. Quy trình phải so sánh từng phần tử $A[i]$ với từng phần tử trong toàn bộ mảng con được sắp xếp $A[1:i-1]$, & do đó $t_i = i$ đối với $i = 2, \ldots, n$. (Quy trình tìm $A[j] > key$ mọi lúc ở dòng 5, vòng lặp & `while` chỉ thoát khi j đạt đến 0.) Lưu ý $\sum_{i=2}^{n} i = (\sum_{i=1}^{n} i) - 1 = \frac{n(n+1)}{2} - 1$ & $\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, tìm trong trường hợp xấu nhất, thời gian chạy của INSERTION-SORT là

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \Big(c_1 + c_2 + c_4 +$$

Có thể biểu thị thời gian chạy trường hợp xấu nhất này là $an^2 + bn + c$ đối với các hằng số a, b, c mà 1 lần nữa phụ thuộc vào chi phí câu lệnh c_k (bây giờ $a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}, b = c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8, c = -(c_2 + c_4 + c_5 + c_8)$). Do đó, thời gian chạy là 1 *hàm bậc hai* của n.

Typically, as in insertion sort, running time of an algorithm is fixed for a given input, although also see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

– Thông thường, giống như trong sắp xếp chèn, thời gian chạy của 1 thuật toán là cố định đối với 1 đầu vào nhất định, mặc dù cũng có 1 số thuật toán "ngẫu nhiên" thú vị mà hành vi của chúng có thể thay đổi ngay cả đối với 1 đầu vào cố định.

Worst-case & average-case analysis. Our analysis of insertion sort looked at both best case, in which input array was already sorted, & worst case, in which input array was reverse sorted. For remainder of this book, though, usually (but not always) concentrate on finding only *worst-case running time*, i.e., longest running time for *any* input of size n. Why? 3 reasons:

– **Phân tích trường hợp xấu nhất & trường hợp trung bình.** Phân tích sắp xếp chèn của chúng tôi xem xét cả trường hợp tốt nhất, trong đó mảng đầu vào đã được sắp xếp, & trường hợp xấu nhất, trong đó mảng đầu vào được sắp xếp ngược. Tuy nhiên, đối với phần còn lại của cuốn sách này, thường (nhưng không phải luôn luôn) tập trung vào việc chỉ tìm *thời gian chạy trường hợp xấu nhất*, tức là thời gian chạy dài nhất cho *bất kỳ* đầu vào nào có kích thước n. Tại sao? 3 lý do:

1. Worst-case running time of an algorithm gives an upper bound on running time for any input. If you know it, then you have a guarantee: algorithm never takes any longer. Need not make some educated guess about running time & hope it never gets much worse. This feature is especially important for real-time computing, in which operations must complete by a deadline.

   – Thời gian chạy tệ nhất của 1 thuật toán cung cấp giới hạn trên về thời gian chạy cho bất kỳ đầu vào nào. Nếu bạn biết điều đó, thì bạn có thể đảm bảo: thuật toán không bao giờ mất nhiều thời gian hơn. Không cần phải đưa ra 1 số phỏng đoán có căn cứ về thời gian chạy & hy vọng nó không bao giờ trở nên tệ hơn nhiều. Tính năng này đặc biệt quan trọng đối với điện toán thời gian thực, trong đó các hoạt động phải hoàn thành trước thời hạn.
2. For some algorithms, worst case occurs fairly often. E.g., in searching a database for a particular piece of information, searching algorithm's worst case often occurs when information is not present in database. In some applications, searches for absent information may be frequent.

   – Đối với 1 số thuật toán, trường hợp xấu nhất xảy ra khá thường xuyên. Ví dụ, khi tìm kiếm 1 thông tin cụ thể trong cơ sở dữ liệu, trường hợp xấu nhất của thuật toán tìm kiếm thường xảy ra khi thông tin không có trong cơ sở dữ liệu. Trong 1 số ứng dụng, việc tìm kiếm thông tin vắng mặt có thể diễn ra thường xuyên.
3. "Average case" is often roughly as bad as worst case. Suppose: run insertion sort on an array of n randomly chosen numbers. How long does it take to determine where in subarray $A[1:i-1]$ to insert element $A[i]$? On average, half elements in $A[1:i-1]$ are less than $A[i]$, & half elements are greater. On average, therefore, $A[i]$ is compared with just half of subarray $A[1:i-1]$, & so t_i is about $\frac{i}{2}$. Resulting average-case running time turns out to be a quadratic function of input size, just like worst-case running time.

   – "Trường hợp trung bình" thường tệ gần bằng trường hợp tệ nhất. Giả sử: chạy sắp xếp chèn trên 1 mảng gồm n số được chọn ngẫu nhiên. Phải mất bao lâu để xác định vị trí trong mảng con $A[1:i-1]$ để chèn phần tử $A[i]$? Trung

bình, 1 nửa phần tử trong $A[1:i-1]$ nhỏ hơn $A[i]$, & 1 nửa phần tử lớn hơn. Do đó, trung bình, $A[i]$ được so sánh với chỉ 1 nửa mảng con $A[1:i-1]$, & do đó t_i bằng khoảng $\frac{i}{2}$. Thời gian chạy trường hợp trung bình thu được hóa ra là 1 hàm bậc hai của kích thước đầu vào, giống như thời gian chạy trường hợp tệ nhất.

In some particular cases, interested in *average-case* running time of an algorithm. See technique of *probabilistic analysis* applied to various algorithms throughout this book. Scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem. Often, assume: all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a *randomized algorithm*, which makes random choices, to allow a probabilistic analysis & yield an *expected* running time. Explore randomized algorithms more in Chap. 5 & in several other subsequent chaps.

– Trong 1 số trường hợp cụ thể, quan tâm đến thời gian chạy *average-case* của 1 thuật toán. Xem kỹ thuật *phân tích xác suất* được áp dụng cho nhiều thuật toán khác nhau trong toàn bộ cuốn sách này. Phạm vi của phân tích trường hợp trung bình bị hạn chế, vì có thể không rõ ràng những gì cấu thành nên đầu vào "trung bình" cho 1 vấn đề cụ thể. Thường thì, hãy giả sử: tất cả các đầu vào có kích thước nhất định đều có khả năng xảy ra như nhau. Trong thực tế, giả định này có thể bị vi phạm, nhưng đôi khi chúng ta có thể sử dụng *thuật toán ngẫu nhiên*, thuật toán này đưa ra các lựa chọn ngẫu nhiên, để cho phép phân tích xác suất & tạo ra thời gian chạy *dự kiến*. Khám phá thêm về các thuật toán ngẫu nhiên trong Chương 5 & trong 1 số chương tiếp theo khác.

Order of growth. In order to ease our analysis of INSERTION-SORT procedure, used some simplifying abstractions. 1st, ignored actual cost of each statement, using constants c_k to represent these costs. Still, best-case & worst-case running times in (2.1)–(2.2) are rather unwieldy. Constants in these expressions give us more detail than we really need. That's why we also expressed best-case running time as $an+b$ for constants a, b that depend on statement costs c_k & why we expressed worst-case running time as an^2+bn+c for constants a, b, c that depend on statement costs. Thus ignored not only actual statement costs, but also abstract costs c_k.

– **Thứ tự tăng trưởng.** Để dễ dàng phân tích quy trình INSERTION-SORT, chúng tôi đã sử dụng 1 số phép trừu tượng đơn giản hóa. Đầu tiên, bỏ qua chi phí thực tế của mỗi câu lệnh, sử dụng hằng số c_k để biểu diễn các chi phí này. Tuy nhiên, thời gian chạy tốt nhất & xấu nhất trong (2.1)–(2.2) khá khó sử dụng. Các hằng số trong các biểu thức này cung cấp cho chúng ta nhiều chi tiết hơn mức chúng ta thực sự cần. Đó là lý do tại sao chúng tôi cũng biểu thị thời gian chạy tốt nhất là $an+b$ cho các hằng số a, b phụ thuộc vào chi phí câu lệnh c_k & tại sao chúng tôi biểu thị thời gian chạy xấu nhất là an^2+bn+c cho các hằng số a, b, c phụ thuộc vào chi phí câu lệnh. Do đó, không chỉ bỏ qua chi phí câu lệnh thực tế mà còn bỏ qua chi phí trừu tượng c_k.

Make 1 more simplifying abstraction: it is *rate of growth*, or *order of growth*, of running time that really interests us. Therefore consider only leading term of a formula (e.g., an^2), since lower-order terms are relatively insignificant for large values of n. Also ignore leading term's constant coefficient, since constant factors are less significant than rate of growth in determining computational efficiency for large inputs. For insertion sort's worst-case running time, when ignore lower-order terms & leading term's constant coefficient, only factor of n^2 from leading term remains. That factor, n^2, is by far most important part of running time. E.g., suppose: an algorithm implemented on a particular machine takes $\frac{n^2}{100}+100n+17$ microseconds on an input of size n. Although coefficients of $\frac{1}{100}$ for n^2 term & 100 for n term differ by 4 orders of magnitude, $\frac{n^2}{100}$ term dominates $100n$ term once n exceeds 10000. Although 10000 might seem large, it is smaller than population of an average town. Many real-world problems have much larger input sizes.

– Tạo thêm 1 phép trừu tượng đơn giản hóa: đó là *tốc độ tăng trưởng*, hoặc *thứ tự tăng trưởng*, của thời gian chạy thực sự khiến chúng ta quan tâm. Do đó, chỉ xem xét số hạng đầu của 1 công thức (ví dụ: an^2), vì các số hạng bậc thấp tương đối không đáng kể đối với các giá trị lớn của n. Ngoài ra, hãy bỏ qua hệ số hằng số của số hạng đầu, vì các hệ số hằng số ít quan trọng hơn tốc độ tăng trưởng trong việc xác định hiệu quả tính toán cho các đầu vào lớn. Đối với thời gian chạy trường hợp xấu nhất của sắp xếp chèn, khi bỏ qua các số hạng bậc thấp & hệ số hằng số của số hạng đầu, chỉ còn lại hệ số n^2 từ số hạng đầu. Hệ số đó, n^2, là phần quan trọng nhất của thời gian chạy. Ví dụ: giả sử: 1 thuật toán được triển khai trên 1 máy cụ thể mất $\frac{n^2}{100}+100n+17$ micro giây trên đầu vào có kích thước n. Mặc dù hệ số của $\frac{1}{100}$ cho số hạng n^2 & 100 cho số hạng n khác nhau 4 bậc độ lớn, số hạng $\frac{n^2}{100}$ chi phối số hạng $100n$ khi n vượt quá 10000. Mặc dù 10000 có vẻ lớn, nhưng nó nhỏ hơn dân số của 1 thị trấn trung bình. Nhiều vấn đề trong thế giới thực có kích thước đầu vào lớn hơn nhiều.

To highlight order of growth of running time, we have a special notation that uses Greek letter Θ. Write: insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared" or just "theta n-squared"). Also write: insertion sort has a best-case running time of $\Theta(n)$ ("theta of n" or "theta n"). For now, think of Θ-notation as saying "roughly proportional when n is large", so that $\Phi(n^2)$ means "roughly proportional to n^2 when n is large" & $\Theta(n)$ means "roughly proportional to n when n is large". Use Θ-notation informally in this chap & define it precisely in Chap. 3.

– Để làm nổi bật thứ tự tăng trưởng của thời gian chạy, chúng ta có 1 ký hiệu đặc biệt sử dụng chữ cái Hy Lạp Θ. Viết: sắp xếp chèn có thời gian chạy trường hợp xấu nhất là $\Theta(n^2)$ (phát âm là "theta của n bình phương" hoặc chỉ là "theta n bình phương"). Cũng viết: sắp xếp chèn có thời gian chạy trường hợp tốt nhất là $\Theta(n)$ ("theta của n" hoặc "theta n"). Bây giờ, hãy nghĩ về ký hiệu Θ như thể hiện "tương xứng khi n lớn", do đó $\Phi(n^2)$ có nghĩa là "tương xứng với n^2 khi n lớn" & $\Theta(n)$ có nghĩa là "tương xứng với n khi n lớn". Sử dụng ký hiệu Θ 1 cách không chính thức trong chương này & định nghĩa chính xác trong Chương. 3.

Usually consider 1 algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors & lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But on large enough inputs, an algorithm whose worst-case running time is $\Theta(n^2)$, e.g., takes less time in worst case than an algorithm whose worst-case

running time is $\Theta(n^3)$. Regardless of constants hidden by Θ-notation, there is always some number, say n_0, s.t. for all input sizes $n \geq n_0$, $\Theta(n^2)$ algorithm beats $\Theta(n^3)$ algorithm in worst case.
– Thông thường coi 1 thuật toán là hiệu quả hơn thuật toán khác nếu thời gian chạy trường hợp xấu nhất của nó có bậc tăng trưởng thấp hơn. Do các yếu tố hằng số & các số hạng bậc thấp hơn, 1 thuật toán có thời gian chạy có bậc tăng trưởng cao hơn có thể mất ít thời gian hơn cho các đầu vào nhỏ so với thuật toán có thời gian chạy có bậc tăng trưởng thấp hơn. Nhưng trên các đầu vào đủ lớn, 1 thuật toán có thời gian chạy trường hợp xấu nhất là $\Theta(n^2)$, ví dụ, mất ít thời gian hơn trong trường hợp xấu nhất so với 1 thuật toán có thời gian chạy trường hợp xấu nhất là $\Theta(n^3)$. Bất kể các hằng số ẩn bởi ký hiệu Θ, luôn có 1 số nào đó, chẳng hạn như n_0, s.t. cho mọi kích thước đầu vào $n \geq n_0$, thuật toán $\Theta(n^2)$ đánh bại thuật toán $\Theta(n^3)$ trong trường hợp xấu nhất.

Problem 8 ([Cor+22], 2.2-1, p. 63). *Express function* $\frac{n^3}{1000} + 100n^2 - 100n + 3$ *in terms of* Θ*-notation.*
– *Biểu diễn hàm số* $\frac{n^3}{1000} + 100n^2 - 100n + 3$ *theo ký hiệu* Θ.

Problem 9 ([Cor+22], 2.2-2, pp. 63–64). *Considering sorting* n *numbers stored in array* $A[1:n]$ *by 1st finding smallest element of* $A[1:n]$ *& exchange it with element in* $A[1]$. *Then find smallest element of* $A[2:n]$, *& exchange it with* $A[2]$. *Then find smallest element of* $A[3:n]$, *& exchange it with* $A[3]$. *Continue in this manner for 1st* $n-1$ *elements of* A. *Write pseudocode for this algorithm, which is known as* selection sort. *What loop invariant does this algorithm maintain? Why does it need to run for only 1st* $n-1$ *elements, rather than* $\forall n$ *elements? Give worst-case running time of selection sort in* Θ*-notation. Is best-case running time any better?*
– *Xét việc sắp xếp* n *số được lưu trữ trong mảng* $A[1:n]$ *bằng cách đầu tiên tìm phần tử nhỏ nhất của* $A[1:n]$ *& hoán đổi nó với phần tử trong* $A[1]$. *Sau đó tìm phần tử nhỏ nhất của* $A[2:n]$, *& hoán đổi nó với* $A[2]$. *Sau đó tìm phần tử nhỏ nhất của* $A[3:n]$, *& hoán đổi nó với* $A[3]$. *Tiếp tục theo cách này cho* $n-1$ *phần tử đầu tiên của* A. *Viết mã giả cho thuật toán này, được gọi là* selection sort. *Thuật toán này duy trì bất biến vòng lặp nào? Tại sao nó chỉ cần chạy cho* $n-1$ *phần tử đầu tiên, thay vì* $\forall n$ *phần tử? Cung cấp thời gian chạy trường hợp xấu nhất của sắp xếp lựa chọn theo ký hiệu* Θ. *Thời gian chạy trường hợp tốt nhất có tốt hơn không?*

Problem 10 ([Cor+22], 2.2-3, p. 64). *Consider linear search again. How many elements of input array need to be checked on average, assuming: element being searched for is equally likely to be any element in array? How about in worst case? Using* Θ*-notation, give average-case & worst-case running times of linear search. Justify answers.*
– *Xem xét lại tìm kiếm tuyến tính. Trung bình cần kiểm tra bao nhiêu phần tử của mảng đầu vào, giả sử: phần tử đang được tìm kiếm có khả năng là bất kỳ phần tử nào trong mảng? Còn trong trường hợp xấu nhất thì sao? Sử dụng ký hiệu* Θ, *đưa ra thời gian chạy trường hợp trung bình & trường hợp xấu nhất của tìm kiếm tuyến tính. Giải thích câu trả lời.*

Problem 11 ([Cor+22], 2.2-4, p. 64). *How can you modify any sorting algorithm to have a good best-case running time?*
– *Làm thế nào bạn có thể sửa đổi bất kỳ thuật toán sắp xếp nào để có thời gian chạy tốt nhất?*

∗ 2.3. Designing algorithms. Can choose from a wide range of algorithm design techniques. Insertion sort uses *incremental* method: for each element $A[i]$, insert it into its proper place in subarray $A[1:i]$, having already sorted subarray $A[1:i-1]$.
– Có thể lựa chọn từ nhiều kỹ thuật thiết kế thuật toán. Sắp xếp chèn sử dụng phương pháp *gia tăng*: đối với mỗi phần tử $A[i]$, chèn nó vào đúng vị trí của nó trong mảng con $A[1:i]$, sau khi đã sắp xếp mảng con $A[1:i-1]$.
This sect examines another design method, known as "divide-&-conquer" (Chap. 4). Use divide-&-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. 1 advantage of using an algorithm that follows divide-&-conquer method: analyzing its running time is often straightforward, using techniques explored in Chap. 4.
– Phần này sẽ xem xét 1 phương pháp thiết kế khác, được gọi là "divide-&-conquer" (Chương 4). Sử dụng divide-&-conquer để thiết kế 1 thuật toán sắp xếp có thời gian chạy trường hợp xấu nhất ít hơn nhiều so với phương pháp sắp xếp chèn. 1 lợi thế của việc sử dụng thuật toán theo phương pháp divide-&-conquer: việc phân tích thời gian chạy của thuật toán này thường rất đơn giản, bằng cách sử dụng các kỹ thuật được khám phá trong Chương 4.

· 2.3.1. Divide-&-conquer method. Many useful algorithms are *recursive* in structure: to solve a given problem, they *recurse* (called themselves) 1 or more times to handle closely related subproblems. These algorithms typically follow *divide-&-conquer* method: they break problem into several subproblems that are similar to original problem but smaller in size, solve subproblems recursively, & then combine these solutions to create a solution to original problem.
– Nhiều thuật toán hữu ích có cấu trúc *đệ quy*: để giải 1 bài toán nhất định, chúng *đệ quy* (gọi chính chúng) 1 hoặc nhiều lần để xử lý các bài toán con có liên quan chặt chẽ. Các thuật toán này thường theo phương pháp *chia-&-chinh phục*: chúng chia bài toán thành nhiều bài toán con tương tự như bài toán gốc nhưng nhỏ hơn về kích thước, giải các bài toán con theo cách đệ quy, & sau đó kết hợp các giải pháp này để tạo ra giải pháp cho bài toán gốc.
In divide-&-conquer method, if problem is small enough – *base case* – you just solve it directly without recursing. Otherwise – *recursive acse* – perform 3 characteristic steps:

1. *Divide* problem into 1 or more subproblems that are smaller instances of same problem.
2. *Conquer* subproblems by solving them recursively.
3. *Combine* subproblem solutions to form a solution to original problem.

– Trong phương pháp chia-&-chinh phục, nếu bài toán đủ nhỏ – *trường hợp cơ sở* – bạn chỉ cần giải quyết trực tiếp mà không cần đệ quy. Nếu không – *đệ quy acse* – thực hiện 3 bước đặc trưng:

1. *Chia* bài toán thành 1 hoặc nhiều bài toán con là các trường hợp nhỏ hơn của cùng 1 bài toán.
2. *Chinh phục* các bài toán con bằng cách giải chúng theo cách đệ quy.
3. *Kết hợp* các giải pháp bài toán con để tạo thành 1 giải pháp cho bài toán ban đầu.

Merge sort algorithm closely follows divide-&-conquer method. In each step, it sorts a subarray $A[p:r]$, starting with entire array $A[1:n]$ & recursing down to smaller & smaller subarrays. Here is how merge sort operates:

1. *Divide* subarray $A[p:r]$ to be sorted into 2 adjacent subarrays, each of half size. To do so, compute midpoint q of $A[p:r]$ (taking average of p, r), & divide $A[p:r]$ into subarrays $A[p:q], A[q+1:r]$.
2. *Conquer* by sorting each of 2 subarrays $A[p:q], A[q+1:r]$ recursively using merge sort.
3. *Combine* by merging 2 sorted subarrays $A[p:q], A[q+1:r]$ back into $A[p:r]$, producing sorted answer.

– Thuật toán *Merge Sort* tuân theo chặt chẽ phương pháp chia-&-chinh phục. Ở mỗi bước, nó sắp xếp 1 mảng con $A[p:r]$, bắt đầu với toàn bộ mảng $A[1:n]$ & đệ quy xuống các mảng con nhỏ hơn & nhỏ hơn. Sau đây là cách sắp xếp hợp nhất hoạt động:

1. *Chia* mảng con $A[p:r]$ để sắp xếp thành 2 mảng con liền kề, mỗi mảng có kích thước bằng 1 nửa. Để thực hiện như vậy, hãy tính điểm giữa q của $A[p:r]$ (lấy trung bình của p, r), & chia $A[p:r]$ thành các mảng con $A[p:q], A[q+1:r]$.
2. *Chinh phục* bằng cách sắp xếp đệ quy từng mảng con $A[p:q], A[q+1:r]$ bằng cách sử dụng sắp xếp hợp nhất.
3. *Kết hợp* bằng cách hợp nhất 2 mảng con đã sắp xếp $A[p:q], A[q+1:r]$ trở lại $A[p:r]$, tạo ra câu trả lời đã sắp xếp.

Recursion "bottoms out" – it reaches base case – when subarray $A[p:r]$ to be sorted has just 1 element, i.e., when p equals r. As noted in initialization argument for INSERTION-SORT's loop invariant, a subarray comprising just a single element is always sorted.
– Đệ quy "chạm đáy" – nó đạt đến trường hợp cơ sở – khi mảng con $A[p:r]$ cần sắp xếp chỉ có 1 phần tử, tức là khi p bằng r. Như đã lưu ý trong đối số khởi tạo cho bất biến vòng lặp INSERTION-SORT, 1 mảng con chỉ bao gồm 1 phần tử duy nhất luôn được sắp xếp.

Key operation of merge sort algorithm occurs in "combine" step, which merges 2 adjacent, sorted subarrays. Merge operation is performed by auxiliary procedure MERGE(A, p, q, r), where A is an array & p, q, r are indices into array s.t. $p \le q < r$. Procedure assumes: adjacent subarrays $A[p:q], A[q+1:r]$ were already recursively sorted. It merges 2 sorted subarrays to form a single sorted subarray that replaces current subarray $A[p:r]$.
– Hoạt động chính của thuật toán sắp xếp trộn xảy ra trong bước "gộp", hợp nhất 2 mảng con liền kề đã được sắp xếp. Hoạt động hợp nhất được thực hiện bởi thủ tục phụ MERGE(A, p, q, r), trong đó A là 1 mảng & p, q, r là các chỉ số vào mảng s.t. $p \le q < r$. Thủ tục giả định: các mảng con liền kề $A[p:q], A[q+1:r]$ đã được sắp xếp đệ quy. Nó hợp nhất 2 mảng con đã được sắp xếp để tạo thành 1 mảng con đã được sắp xếp duy nhất thay thế mảng con hiện tại $A[p:r]$.
p. 66+++

- 3. Characterizing Running Times.
- 4. Divide-&-Conquer.
- 5. Probabilistic Analysis & Randomized Algorithms.

- II. SORTING & ORDER STATISTICS.
  - Introduction.
  - 6. Heapsort.
  - 7. Quicksort.
  - 8. Sorting in Linear Time.
  - 9. Medians & Other Statistics.

- III. DATA STRUCTURES.
  - Introduction.
  - 10. Elementary Data Structures.
  - 11. Hash Tables.
  - 12. Binary Search Trees.
  - 13. Red-Black Trees.

- IV. ADVANCED DESIGN & ANALYSIS TECHNIQUES.
  - Introduction. This part covers 3 important techniques used in designing & analyzing efficient algorithms: dynamic programming (Chap. 14), greedy algorithms (Chap. 15), & amortized analysis (Chap. 16). Earlier parts have presented other widely applicable techniques, e.g. divide-&-conquer, randomization, & how to solve recurrences. Techniques in this part are somewhat more sophisticated, but will be able to use them solve many computational problems. Themes introduced in this part will recur later in this book.

    Dynamic programming typically applies to optimization problems in which make a set of choices in order to arrive at an optimal solution, each choice generates subproblems of same form as original problem, & same subproblems arise repeatedly. Key strategy: store solution to each such subproblem rather than recompute it. Chap. 14 shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

    Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which you make a set of choices in order to arrive at an optimal solution. Idea of a greedy algorithm: make each choice in a locally optimal

manner, resulting in a faster algorithm than you get with dynamic programming. Chap. 15 will help you determine when greedy approach works.

Technique of amortized analysis (phân tích khấu hao) applies to certain algorithms that perform a sequence of similar operations. Instead of bounding cost of sequence of operations by bounding actual cost of each operation separately, an amortized analysis provides a worst-case bound on actual cost of entire sequence. 1 advantage of this approach: although some operations might be expensive, many others might be cheap. Can use amortized analysis when designing algorithms, since design of an algorithm & analysis of running time are often closely intertwined. Chap. 16 introduces 3 ways to perform an amortized analysis of an algorithm.

- 14. Dynamic Programming. Dynamic programming, like divide-&-conquer method, solves problems by combining solutions to subproblems. ("Programming" in this context refers to a tabular method, not to writing computer code.) As saw in Chaps. 2 & 4, divide-&-conquer algorithms partition problem into disjoint subproblems, solve subproblems recursively, & then combine their solutions to solve original problem. In contrast, dynamic programming applies when subproblems overlap – i.e., when subproblems share subsubproblems. In this context, a divide-&-conquer algorithm does more work than necessary, repeatedly solving common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once & then saves its answer in a table, thereby avoiding work of recomputing answer every time it solves each subsubproblem.

  Dynamic programming typically applies to *optimization problems*. Such problems can have many possible solutions. Each solution has a value, & want to find a solution with optimal (minimum or maximum) value. Call such a solution *an* optimal solution to problem, as opposed to *the* optimal solution, since there may be several solutions that achieve optimal value.

  To develop a dynamic-programming algorithm, follow a sequence of 4 steps:

  1. Characterize structure of an optimal solution.
  2. Recursively define value of an optimal solution.
  3. Compute value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

  Steps 1–3 form basis of a dynamic-programming solution to a problem. If need only value of an optimal solution, & not solution itself, then can omit step 4. When do perform step 4, it often pays to maintain additional information during step 3 so that can easily construct an optimal solution.

  Sects that follow use dynamic-programming method to solve some optimization problems. Sects. 14.1 examines problem of cutting a rod into rods of smaller length in a way that maximizes their total value. Sect. 14.2 shows how to multiply a chain of matrices while performing fewest total scalar multiplications. Given these examples of dynamic programming, Sect. 14.3 discusses 2 key characteristics that a problem must have for dynamic programming to be a viable solution technique. Sect. 14.4 then shows how to find longest common subsequence of 2 sequences via dynamic programming. Finally, Sect. 14.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

  - 14.1. Rod cutting. 1st example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods & cuts them into shorter rods, which it then sells. Each cut is free. Management of Serling Enterprises wants to know best way to cut up rods.

    Serling Enterprises has a table giving, for $i = 1, 2, \ldots$, price p_i in dollars that they charge for a rod of length i inches. Length of each rod in inches is always an integer. Fig. 14.1: A sample price table for rods. Each rod of length i inches earns company p_i dollars of revenue. gives a sample price table.

    Rod-cutting problem is following. Given a rod of length n inches & a table of prices p_i for $i = 1, \ldots, n$, determine maximum revenue r_n obtainable by cutting up rod & selling pieces. If price p_n for a rod of length n is large enough, an optimal solution might require no cutting at all.

    Consider case when $n = 4$. Fig. 14.2: 8 possible ways of cutting a rod of length 4. Above each piece is value of that piece, according to sample price chart of Fig. 14.1. Optimal strategy is part (c) – cutting rod into 2 pieces of length 2 – which has total value 10. shows all ways to cut up a rod of 4 inches in length, including way with no cuts at all. Cutting a 4-inch rod into 2 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

    Serling Enterprises can cut up a rod of length n in 2^{n-1} different ways, since they have an independent option of cutting, or not cutting, at distance i inches from left end, for $i = 1, \ldots, n-1$ [If pieces are required to be cut in order of monotonically increasing size, there are fewer ways to consider. For $n = 4$, only 5 such ways are possible: Number of ways is called *partition function*, which is approximately equal to $\frac{e^{\pi\sqrt{\frac{2n}{3}}}}{4n\sqrt{3}}$. This quantity is $< 2^{n-1}$, but still much greater than any polynomial in n. Won't pursue this line of inquiry further, however.] Denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates: a rod of length 7 is cut into 3 pieces – 2 of length 2 & 1 of length 3. If an optimal solution cuts rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition $n = \sum_{j=1}^{k} i_j$ of rod into pieces of lengths $i_1, \ldots, i_k$ provides maximum corresponding revenue $r_n = \sum_{j=1}^{k} p_{i_j}$.

    For sample problem in Fig. 14.1, can determine optimal revenue figures r_i, for $i = 1, \ldots, 10$, by inspection, with corresponding optimal decompositions [value of $r_1, \ldots, r_{10}$].

    More generally, can express values r_n for $n \ge 1$ in terms of optimal revenues from shorter rods: (14.1)

    $$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}.$$

    1st argument p_n corresponds to making no cuts at all & selling rod of length n as is. Other $n-1$ arguments to max correspond to maximum revenue obtained by making an initial cut of rod into 2 pieces of size $i, n-i$, for each $i = 1, \ldots, n-1$,

& then optimally cutting up those pieces further, obtaining revenues r_i, r_{n-i} from those 2 pieces. Since don't know ahead of time which value of i optimizes revenue, have to consider all possible values for i & pick the one that maximizes revenue. Also have option of picking no i at all if greatest revenue comes from selling rod uncut.

To solve original problem of size n, solve smaller problems of same type. Once make 1st cut, 2 resulting pieces form independent instances of rod-cutting problem. Overall optimal solution incorporates optimal solutions to 2 resulting subproblems, maximizing revenue from each of those 2 pieces. Say: rod-cutting problem exhibits *optimal substructure*: optimal solutions to a problem incorporate optimal solutions to related subproblems, which you may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for rod-cutting problem, view a decomposition as consisting of a 1st piece of length i cut off left-hand end, & then a right-hand remainder of length $n-i$. Only remainder, & not 1st piece, may be further divided. Think of every decomposition of a length-n rod in this way: as a 1st piece followed by some decomposition of remainder. Then can express solution with no cuts at all by saying: 1st piece has size $i = n$ & revenue p_n & remainder has size 0 with corresponding revenue $r_0 = 0$. Thus obtain following simpler version of equation (14.1): (14.2)

$$r_n = \max\{p_i + r_{n_i} : 1 \le i \le n\}.$$

In this formulation, an optimal solution embodies solution to only *1* related subproblem – remainder – rather than 2.

· **Recursive top-down implementation.** CUT-ROD procedure implements computation implicit in (14.2) in a straightforward, top-down, recursive manner. It takes as input an array $p[1 : n]$ of prices & an integer n, & it returns maximum revenue possible for a rod of length n. For length $n = 0$, no revenue is possible, & so CUT-ROD returns 0 in line 2. Line 3 initializes maximum revenue q to $-\infty$, so that `for` loop in lines 4–5 correctly computes $q = \max\{p_i + CUT - ROD(p, n-i) : 1 \le i \le n\}$. Line 6 then returns this value. A simple induction on n proves: this answer = desired answer r_n using (14.2). `[CUT-ROD`(p, n) `algorithm]`.

If code up CUT-ROD in favorite programming language & run it on your computer, find: once input size becomes moderately large, your program takes a long time to run. For $n = 40$, your program may take several minutes & possibly > 1 hour. For large values of n, also discover: each time increase n by 1, your program's running time approximately doubles.

Why is CUT-ROD so inefficient? Problem: CUT-ROD calls itself recursively over & over again with same parameter values, i.e., it solves same subproblems repeatedly. Fig. 14.3: Recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives size n of corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ & leaving a remaining subproblem of size t. A path from root to a leaf corresponds to 1 of 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes & 2^{n-1} leaves. shows a recursion tree demonstrating what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD$(p, n-i)$ for $i = 1, \ldots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \ldots, n-1$. When this process unfolds recursively, amount of work done, as a function of n, grows explosively.

To analyze running time of CUT-ROD, let $T(n)$ denote total number of calls made to CUT-ROD(p, n) for a particular value of n. This expression equals number of nodes in a subtree whose root is labeled n in recursion tree. Count includes initial call at its root. Thus, $T(0) = 1$ & (14.3)

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j).$$

Initial 1 is for call at root, & term $T(j)$ counts number of calls (including recursive calls) due to call CUT-ROD$(p, n-i)$, where $j = n - i$. Exercise 14.1-1: $T(n) = 2^n$ & so running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all possible ways of cutting up a rod of length n. How many ways are there? A rod of length n has $n - 1$ potential locations to cut. Each possible way to cut up rod makes a cut at some subset of these $n-1$ locations, including empty set, which makes for no cuts. Viewing each cut location as a distinct member of a set of $n - 1$ elements, can see: there are 2^{n-1} subsets. Each leaf in recursion tree of Fig. 14.3 corresponds to 1 possible way to cut up rod. Hence, recursion tree has 2^{n-1} leaves. Labels on simple path from root to a leaf give sizes of each remaining right-hand piece before making each cut. I.e., labels give corresponding cut points, measured from right-hand end of rod.

· **Using dynamic programming for optimal rod cutting.**

- 15. Greedy Algorithms.
- 16. Amortized Analysis.

- V. ADVANCED DATA STRUCTURES.
  - Introduction.
  - 17. Augmenting Data Structures.
  - 18. B-Trees.
  - 19. Data Structures for Disjoint Sets.

- VI. GRAPH ALGORITHMS.

- Introduction. Graphs problems pervade CS, & algorithms for working with them are fundamental to the field. Hundreds of interesting computational problems are couched in terms of graphs. This part touches on a few of the more significant ones.

  – Các vấn đề về đồ thị tràn ngập trong CS, & các thuật toán để làm việc với chúng là nền tảng cho lĩnh vực này. Hàng trăm vấn đề tính toán thú vị được trình bày dưới dạng đồ thị. Phần này đề cập đến 1 số vấn đề quan trọng hơn.

  Chap. 20 shows how to represent a graph in a computer & then discusses algorithms based on searching a graph using either BFS or DFS. Chap gives 2 applications of DFS: topologically sorting a directed acyclic graph & decomposing a directed graph into its strongly connected components.

  – Chương 20 trình bày cách biểu diễn đồ thị trong máy tính & sau đó thảo luận về các thuật toán dựa trên việc tìm kiếm đồ thị bằng BFS hoặc DFS. Chương này đưa ra 2 ứng dụng của DFS: sắp xếp tôpô 1 đồ thị có hướng không có chu trình & phân tích 1 đồ thị có hướng thành các thành phần liên thông mạnh của nó.

  Chap. 21 describes how to compute a minimum-weight spanning tree of a graph: least-weight way of connecting all of vertices together when each edge has an associated weight. Algorithms for computing minimum spanning trees serve as good examples of greedy algorithms (Chap. 15).

  – Chương 21 mô tả cách tính cây khung có trọng số tối thiểu của đồ thị: cách có trọng số tối thiểu để kết nối tất cả các đỉnh với nhau khi mỗi cạnh có trọng số liên quan. Các thuật toán để tính cây khung tối thiểu đóng vai trò là ví dụ tốt về thuật toán tham lam (Chương 15).

  Chaps. 22–23 consider how to compute shortest paths between vertices when each edge has an associated length or "weight". Chap. 22 shows how to find shortest paths from a given source vertex to all other vertices, & Chap. 23 examines methods to compute shortest paths between every pair of vertices.

  – Chương 22–23 xem xét cách tính đường đi ngắn nhất giữa các đỉnh khi mỗi cạnh có chiều dài hoặc "trọng số" liên quan. Chương 22 cho thấy cách tìm đường đi ngắn nhất từ 1 đỉnh nguồn cho trước đến tất cả các đỉnh khác, & Chương 23 kiểm tra các phương pháp tính đường đi ngắn nhất giữa mọi cặp đỉnh.

  Chap. 24 shows how to compute a maximum flow of material in a flow network, which is a directed graph having a specified source vertex of material, a specified sink vertex, & specified capacities for amount of material that can traverse each directed edge. This general problem arises in many forms, & a good algorithm for computing maximum flows can help solve a variety of related problems efficiently.

  – Chương 24 trình bày cách tính toán lưu lượng vật liệu cực đại trong mạng lưu lượng, là đồ thị có hướng có đỉnh nguồn vật liệu xác định, đỉnh chìm xác định, & dung lượng xác định cho lượng vật liệu có thể đi qua mỗi cạnh có hướng. Vấn đề chung này phát sinh ở nhiều dạng, & 1 thuật toán tốt để tính lưu lượng cực đại có thể giúp giải quyết hiệu quả nhiều vấn đề liên quan.

  Chap. 25 explores matchings in bipartite graphs: methods for pairing up vertices that are partitioned into 2 sets by selecting edges that go between sets. Bipartite-matching problems model several situations that arise in real world. Chap examines how to find a matching of maximum cardinality; "stable-marriage problem", which has highly practical application of matching medical residents to hospitals; & assignment problems, which maximize total weight of a bipartite matching.

  – Chương 25 khám phá các phép ghép trong đồ thị hai phần: các phương pháp ghép các đỉnh được phân vùng thành 2 tập bằng cách chọn các cạnh nằm giữa các tập. Các bài toán ghép hai phần mô hình hóa 1 số tình huống phát sinh trong thế giới thực. Chương này xem xét cách tìm phép ghép có số lượng tối đa; "bài toán hôn nhân ổn định", có ứng dụng thực tế cao trong việc ghép các bác sĩ nội trú với các bệnh viện; & các bài toán gán, giúp tối đa hóa tổng trọng số của phép ghép 2 phần.

  When characterize running time of a graph algorithm on a given graph $G = (V, E)$, usually measure size of input in terms of number of vertices $|V|$ & number of edges $|E|$ of graph. I.e., denote size of input with 2 parameters, not just 1. Adopt a common notational convention for these parameters. Inside asymptotic notation (e.g. O-notation or Θ-notation), & *only* inside such notation, symbol V denotes $|V|$ & symbol E denotes $|E|$. E.g., might say, "algorithm runs in $O(VE)$ time", i.e., algorithm runs in $O(|V||E|)$ time. This convention makes running-time formulas easier to read, without risk of ambiguity.

  – Khi mô tả thời gian chạy của 1 thuật toán đồ thị trên 1 đồ thị cho trước $G = (V, E)$, thường đo kích thước của đầu vào theo số đỉnh $|V|$ & số cạnh $|E|$ của đồ thị. Tức là, biểu thị kích thước của đầu vào bằng 2 tham số, không chỉ 1. Áp dụng 1 quy ước ký hiệu chung cho các tham số này. Bên trong ký hiệu tiệm cận (ví dụ: ký hiệu O hoặc ký hiệu Θ), & *only* bên trong ký hiệu như vậy, ký hiệu V biểu thị $|V|$ & ký hiệu E biểu thị $|E|$. Ví dụ, có thể nói, "thuật toán chạy trong thời gian $O(VE)$", tức là thuật toán chạy trong thời gian $O(|V||E|)$. Quy ước này giúp các công thức thời gian chạy dễ đọc hơn, không có nguy cơ mơ hồ.

  Another convention adopted appears in pseudocode. Denote vertex set of a graph G by $G.V$ & its edge set by $G.E$. I.e., pseudocode views vertex & edge sets as attributes of a graph.

  – Một quy ước khác được áp dụng xuất hiện trong mã giả. Ký hiệu tập đỉnh của đồ thị G bởi $G.V$ & tập cạnh của nó bởi $G.E$. I.e.,, mã giả xem tập đỉnh & cạnh là các thuộc tính của đồ thị.

- 20. Elementary Graph Algorithms. This chap presents methods for representing a graph & for searching a graph. Searching a graph means systematically following edges of graph so as to visit vertices of graph. A graph-searching algorithm can discover much about structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at heart of field of graph algorithms.

  – Chương này trình bày các phương pháp để biểu diễn 1 đồ thị & để tìm kiếm 1 đồ thị. Tìm kiếm 1 đồ thị có nghĩa là theo dõi 1 cách có hệ thống các cạnh của đồ thị để thăm các đỉnh của đồ thị. Một thuật toán tìm kiếm đồ thị có thể khám phá

nhiều về cấu trúc của 1 đồ thị. Nhiều thuật toán bắt đầu bằng cách tìm kiếm đồ thị đầu vào của chúng để có được thông tin cấu trúc này. Một số thuật toán đồ thị khác giải thích chi tiết về tìm kiếm đồ thị cơ bản. Các kỹ thuật để tìm kiếm 1 đồ thị nằm ở trung tâm của lĩnh vực thuật toán đồ thị.

Sect. 20.1 discusses 2 most common computational representations of graphs: as adjacency lists & as adjacency matrices. Sect. 20.2 presents a simple graph-searching algorithm called BFS & shows how to create a breadth-1st tree. Sect. 20.3 presents depth-1st search & proves some standard results about order in which DFS visits vertices. Sect. 20.4 provides our 1st real application of DFS: topological sorting a directed acyclic graph. A 2nd application of DFS, finding strongly connected components of a directed graph, is topic of Sect. 20.5.

– Mục 20.1 thảo luận về 2 biểu diễn tính toán phổ biến nhất của đồ thị: dưới dạng danh sách kề & dưới dạng ma trận kề. Mục 20.2 trình bày 1 thuật toán tìm kiếm đồ thị đơn giản có tên là BFS & cho thấy cách tạo cây theo chiều rộng 1. Mục 20.3 trình bày tìm kiếm theo chiều sâu 1 & chứng minh 1 số kết quả chuẩn về thứ tự mà DFS duyệt các đỉnh. Mục 20.4 cung cấp ứng dụng thực tế đầu tiên của chúng tôi về DFS: sắp xếp tôpô cho đồ thị có hướng phi chu trình. Ứng dụng thứ 2 của DFS, tìm các thành phần liên thông mạnh của đồ thị có hướng, là chủ đề của Mục 20.5.

- 21. Minimum Spanning Trees. Can choose between 2 standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed & undirected graphs. Because adjacency-list representation provides a compact way to represent *sparse* graphs – those for which $|E|$ is much less than $|V|^2$ – it is usually method of choice. Most of graph algorithms presented in this book assume: an input graph is represented in adjacency-list form. Might prefer an adjacency-matrix representation, however, when graph is *dense* – $|E|$ is close to $|V|^2$ – or when you need to be able to tell quickly whether there is an edge connecting 2 given vertices. E.g., 2 of all-pairs shortest-paths algorithms presented in Chap. 23 assume: their input graphs are represented by adjacency matrices.

  – Cây khung nhỏ nhất. Có thể chọn giữa 2 cách chuẩn để biểu diễn đồ thị $G = (V, E)$: dưới dạng tập hợp các danh sách kề hoặc dưới dạng ma trận kề. Cả hai cách đều áp dụng cho cả đồ thị có hướng & vô hướng. Bởi vì biểu diễn danh sách kề cung cấp 1 cách gọn nhẹ để biểu diễn đồ thị *thưa* – đồ thị mà $|E|$ nhỏ hơn nhiều so với $|V|^2$ – nên đây thường là phương pháp được lựa chọn. Hầu hết các thuật toán đồ thị được trình bày trong cuốn sách này đều giả định: đồ thị đầu vào được biểu diễn dưới dạng danh sách kề. Tuy nhiên, có thể thích biểu diễn ma trận kề hơn khi đồ thị *dày đặc* – $|E|$ gần với $|V|^2$ – hoặc khi bạn cần có khả năng nhanh chóng biết liệu có cạnh nào kết nối 2 đỉnh đã cho hay không. Ví dụ: 2 trong số các thuật toán đường đi ngắn nhất mọi cặp được trình bày trong Chương 23 giả định: đồ thị đầu vào của chúng được biểu diễn bằng ma trận kề.

  p. 720+++
- 22. Single-Source Shortest Paths.
- 23. All-Pairs Shortest Paths.
- 24. Maximum Flow.
- 25. Matching in Bipartite Graphs.

- VII. SELECTED TOPICS.
  - Introduction.
  - 26. Parallel Algorithms.
  - 27. Online Algorithms.
  - 28. Matrix Operations.
  - 29. Linear Programming.
  - 30. Polynomials & FFT.
  - 31. Number-Theoretic Algorithms.
  - 32. String Matching.
  - 33. Machine-Learning Algorithms.
  - 34. NP-Completeness.
  - 35. Approximation Algorithms.
- VIII. APPENDIX: MATHEMATICAL BACKGROUND.
  - Introduction.
  - A. Summations.
  - B. Sets, etc.
  - C. Counting & Probability.
  - D. Matrices.

2.2 [Knu98]. Donald Erwin Knuth. The Art of Computer Programming. Vol. 3: Sorting & Searching. 2e

- Preface. "Cookery is become an art, a noble science, cooks are gentlemen." – Titus Livius, *Ab Urbe Condita* XXXIX.vi (Robert Burton, *Anatomy of Melancholy* – Giải phẫu của sự u sầu 1.2.2.2)

  This book forms a natural sequel to material on information structures in Chap. 2 of Vol. 1, because it adds concept of linearly ordered data to other basic structural ideas.

  – Cuốn sách này là phần tiếp theo tự nhiên của tài liệu về cấu trúc thông tin trong Chương 2 của Tập 1, vì nó bổ sung khái niệm dữ liệu được sắp xếp tuyến tính vào các ý tưởng cấu trúc cơ bản khác.

  Title "Sorting & Searching" may sound as if this book is only for those systems programmers who are concerned with preparation of general-purpose sorting routines or applications to information retrieval. But in fact area of sorting & searching provides an ideal framework for discussing a wide variety of important general issues:

  - How are good algorithms discovered?
  - How can efficiency of algorithms be analyzed mathematically?
  - How can a person choose rationally between different algorithms for same task?
  - In what senses can algorithms be proved "best possible"?
  - How does theory of computing interact with practical considerations?
  - How can external memories like tapes, drums, or disks be used efficiently with large database?

  Indeed, Knuth believe: virtually *every* important aspect of programming arises somewhere in context of sorting or searching!

  – Tiêu đề "Sắp xếp & Tìm kiếm" có vẻ như cuốn sách này chỉ dành cho những lập trình viên hệ thống quan tâm đến việc chuẩn bị các quy trình sắp xếp mục đích chung hoặc các ứng dụng để truy xuất thông tin. Nhưng trên thực tế, lĩnh vực sắp xếp & tìm kiếm cung cấp 1 khuôn khổ lý tưởng để thảo luận về nhiều vấn đề chung quan trọng:

  - Các thuật toán tốt được phát hiện như thế nào?
  - Hiệu quả của các thuật toán có thể được phân tích về mặt toán học như thế nào?
  - Làm thế nào 1 người có thể lựa chọn hợp lý giữa các thuật toán khác nhau cho cùng 1 nhiệm vụ?
  - Theo nghĩa nào thì các thuật toán có thể được chứng minh là "tốt nhất có thể"?
  - Lý thuyết điện toán tương tác với các cân nhắc thực tế như thế nào?
  - Làm thế nào để sử dụng hiệu quả các bộ nhớ ngoài như băng, trống hoặc đĩa với cơ sở dữ liệu lớn?

  Thật vậy, Knuth tin rằng: hầu như *mọi* khía cạnh quan trọng của lập trình đều phát sinh ở đâu đó trong bối cảnh sắp xếp hoặc tìm kiếm!

  This volume comprises Chaps. 5–6 of complete series. Chap. 5 is concerned with sorting into order; this is a large subset that has been divided chiefly into 2 parts, internal sorting & external sorting. There also are supplementary sects, which develop auxiliary theories about permutations (Sect. 5.1) & about optimum techniques for sorting (Sect. 5.3). Chap. 6 deals with problem of searching for specified items in tables or files; this is subdivided into methods that search sequentially, or by comparison of keys, or by digital properties, or by hashing, & then more difficult problem of secondary key retrieval is considered. There is a surprising amount of interplay between both chaps, with strong analogies typing topics together. 2 important varieties of information structures are also discussed, in addition to those considered in Chap. 2, namely priority queues (Sect. 5.2.3) & linear lists represented as balanced trees (Sect. 6.2.3).

  – Tập này bao gồm các Chương 5-6 của bộ sách hoàn chỉnh. Chương 5 đề cập đến việc sắp xếp theo thứ tự; đây là 1 tập hợp con lớn được chia chủ yếu thành 2 phần, sắp xếp nội bộ & sắp xếp ngoài. Ngoài ra còn có các giáo phái bổ sung, phát triển các lý thuyết phụ trợ về hoán vị (Mục 5.1) & về các kỹ thuật tối ưu để sắp xếp (Mục 5.3). Chương 6 đề cập đến vấn đề tìm kiếm các mục đã chỉ định trong các bảng hoặc tệp; điều này được chia thành các phương pháp tìm kiếm theo trình tự, hoặc bằng cách so sánh các khóa, hoặc bằng các thuộc tính kỹ thuật số, hoặc bằng cách băm, & sau đó là vấn đề khó hơn về việc truy xuất khóa thứ cấp được xem xét. Có 1 lượng tương tác đáng ngạc nhiên giữa cả hai chương, với các phép loại suy mạnh mẽ khi gõ các chủ đề lại với nhau. 2 loại cấu trúc thông tin quan trọng cũng được thảo luận, ngoài các loại được xem xét trong Chương 2, cụ thể là hàng đợi ưu tiên (Mục 5.2.3) & danh sách tuyến tính được biểu diễn dưới dạng cây cân bằng (Mục 6.2.3).

  Like Vols. 1–2, this book includes a lot of material that does not appear in other publications. Many people have kindly written to me about their ideas, or spoken to me about them, & I hope I have not distorted material too badly when I have presented it in my own words.

  – Giống như Tập 1-2, cuốn sách này bao gồm nhiều tài liệu không xuất hiện trong các ấn phẩm khác. Nhiều người đã tử tế viết thư cho tôi về ý tưởng của họ, hoặc nói chuyện với tôi về chúng, & Tôi hy vọng tôi không bóp méo tài liệu quá tệ khi tôi trình bày theo lời của riêng tôi.

  I have not had time to search patent literature systematically; indeed, I decry current tendency to seek patents on algorithms (Sect. 5.4.5). If somebody sends me a copy of a relevant patent not presently cited in this book, I will dutifully refer to it

in future editions. However, I want to encourage people to continue centuries-old mathematical tradition of putting newly discovered algorithms into public domain. There are better ways to earn a living than to prevent other people from making use of one's contributions to CS.

– Tôi không có thời gian để tìm kiếm tài liệu về bằng sáng chế 1 cách có hệ thống; thực ra, tôi lên án xu hướng hiện nay là tìm kiếm bằng sáng chế về thuật toán (Mục 5.4.5). Nếu ai đó gửi cho tôi 1 bản sao của bằng sáng chế có liên quan hiện không được trích dẫn trong cuốn sách này, tôi sẽ tận tụy tham khảo nó trong các phiên bản sau. Tuy nhiên, tôi muốn khuyến khích mọi người tiếp tục truyền thống toán học đã có từ nhiều thế kỷ là đưa các thuật toán mới được phát hiện vào phạm vi công cộng. Có nhiều cách tốt hơn để kiếm sống hơn là ngăn cản người khác sử dụng những đóng góp của mình cho khoa học máy tính.

Before I retired from teaching, I used this book as a text for a student's 2nd course in data structures, at junior-to-graduate level, omitting most of mathematical material. Also used mathematical portions of this book as basis for graduate-level courses in analysis of algorithms, emphasizing especially Sects. 5.1, 5.2.2, 6.3, & 6.4. A graduate-level course on concrete computational complexity could also be based on Sects. 5.3, & 5.4.4, together with Sects. 4.3.3, 4.6.3, & 4.6.4 of Volume 2.

– Trước khi nghỉ hưu, tôi đã sử dụng cuốn sách này làm giáo trình cho khóa học thứ 2 của sinh viên về cấu trúc dữ liệu, ở trình độ từ cơ sở đến sau đại học, bỏ qua hầu hết các tài liệu toán học. Tôi cũng sử dụng các phần toán học của cuốn sách này làm cơ sở cho các khóa học sau đại học về phân tích thuật toán, đặc biệt nhấn mạnh vào Mục 5.1, 5.2.2, 6.3, & 6.4. Một khóa học sau đại học về độ phức tạp tính toán cụ thể cũng có thể dựa trên Mục 5.3, & 5.4.4, cùng với Mục 4.3.3, 4.6.3, & 4.6.4 của Tập 2.

For most part this book is self-contained, except for occasional discussions relating to `MIX` computer explained in Vol. 1. Appendix B contains a summary of mathematical notations used, some of which are a little different from those found in traditional mathematics books.

– Phần lớn cuốn sách này là nội dung độc lập, ngoại trừ 1 số cuộc thảo luận thỉnh thoảng liên quan đến máy tính `MIX` được giải thích trong Tập 1. Phụ lục B tóm tắt các ký hiệu toán học được sử dụng, 1 số trong đó hơi khác so với những ký hiệu có trong các sách toán học truyền thống.

- Preface to 2e. This new edition matches 3es of Vols. 1–2, in which I have been able to celebrate completion of TEX & METAFONT by applying those systems to publications they were designed for.

  Conversion to electronic format has given me opportunity to go over every word of text & every punctuation mark. Have tried to retain youthful exuberance of original sentences while perhaps adding some more mature judgment. Dozens of new exercises have been added; dozens of old exercises have been given new & improved answers. Changes appear everywhere, but most significantly in Sects. 5.1.4 (about permutations & tableaux), 5.3 (about optimum sorting), 5.4.9 (about disk sorting), 6.2.2 (about entropy), 6.4 (about universal hashing), & 6.5 (about multidimensional trees & tries).

  – Việc chuyển đổi sang định dạng điện tử đã cho tôi cơ hội xem xét lại từng từ trong văn bản & từng dấu câu. Đã cố gắng giữ lại sự phấn khích trẻ trung của các câu gốc trong khi có lẽ thêm 1 số phán đoán chín chắn hơn. Hàng chục bài tập mới đã được thêm vào; hàng chục bài tập cũ đã được đưa ra & câu trả lời mới được cải thiện. Những thay đổi xuất hiện ở khắp mọi nơi, nhưng đáng kể nhất là ở các Mục. 5.1.4 (về hoán vị & bảng), 5.3 (về sắp xếp tối ưu), 5.4.9 (về sắp xếp đĩa), 6.2.2 (về entropy), 6.4 (về băm phổ quát), & 6.5 (về cây đa chiều & thử).

  The Art of Computer Programming is, however, still a work in progress. Researching on sorting & searching continues to grow at a phenomenal rate. Therefore some parts of this book are headed by an "under construction" icon, to apologize for fact: material is not up-to-date. E.g., if I were teaching an undergraduate class on data structures today, I would surely discuss randomized structures e.g. treaps at some length; but at present, only able to cite principal papers on subject, & to announce plans for a future Sect. 6.2.5. My files are bursting with important material that I plan to include in final, glorious, 3e of Vol. 3, perhaps 17 years from now. But I must finish Vols. 4–5 1st, & I do not want to delay their publication any more than absolutely necessary.

  – Tuy nhiên, *Nghệ thuật lập trình máy tính* vẫn đang trong quá trình hoàn thiện. Nghiên cứu về sắp xếp & tìm kiếm tiếp tục phát triển với tốc độ phi thường. Do đó, 1 số phần của cuốn sách này được đánh dấu bằng biểu tượng "đang xây dựng", để xin lỗi vì thực tế: tài liệu không được cập nhật. Ví dụ, nếu tôi đang giảng dạy 1 lớp đại học về cấu trúc dữ liệu ngày nay, tôi chắc chắn sẽ thảo luận về các cấu trúc ngẫu nhiên, ví dụ như treaps ở 1 số phần; nhưng hiện tại, chỉ có thể trích dẫn các bài báo chính về chủ đề này, & để công bố kế hoạch cho Phần 6.2.5 trong tương lai. Các tệp của tôi đang tràn ngập tài liệu quan trọng mà tôi dự định đưa vào bản cuối cùng, tuyệt vời, 3e của Tập 3, có lẽ là 17 năm nữa. Nhưng tôi phải hoàn thành Tập 4-5 trước, & Tôi không muốn trì hoãn việc xuất bản chúng lâu hơn mức cần thiết.

  > "There are certain common Privileges of a Writer, the Benefit whereof, I hope, there will be no Reason to doubt; Particularly, that where I am not understood, it shall be concluded, that something very useful & profound is coucht underneath." – Jonathan Swift, *Tale of a Tub*, Preface (1704) – Có 1 số Quyền lợi chung của 1 Nhà văn, mà tôi hy vọng rằng sẽ không có Lý do gì để nghi ngờ; Đặc biệt, khi tôi không được hiểu, có thể kết luận rằng có điều gì đó rất hữu ích & sâu sắc ẩn chứa bên dưới.

- Notes on Exercises. Exercises in this set of books have been designed for self-study as well as for classroom study. If is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying information to specific problems & thereby being encouraged to think about what has been read. Furthermore, we all learn best things that we have discovered

for ourselves. Therefore exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible & to select problems that are enjoyable as well as instructive.

– Ghi chú về Bài tập. Các bài tập trong bộ sách này được thiết kế để tự học cũng như để học trên lớp. Thật khó, nếu không muốn nói là không thể, đối với bất kỳ ai muốn học 1 môn học chỉ bằng cách đọc về môn học đó, mà không áp dụng thông tin vào các bài toán cụ thể & do đó được khuyến khích suy nghĩ về những gì đã đọc. Hơn nữa, tất cả chúng ta đều học được những điều tốt nhất mà chúng ta tự khám phá ra. Do đó, các bài tập đóng vai trò chính trong công việc này; chúng tôi đã nỗ lực hết sức để giữ cho chúng mang tính thông tin nhất có thể & để chọn các bài toán vừa thú vị vừa mang tính hướng dẫn.

- Chap. 5: Sorting.

  - "There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in introduction of a new order of things." – Niccolò Machiavelli, *The Prince* (1513)

    – Không có gì khó khăn hơn để thực hiện, nguy hiểm hơn để tiến hành, hoặc không chắc chắn hơn về thành công, hơn là việc dẫn đầu trong việc giới thiệu 1 trật tự mới.

  - "But you can't look up all those license numbers in time," Drake objected. "We don't have to, Paul. We merely arrange a list & look for duplications." – Perry Mason, in *he Case of the Angry Mourner* (1951)

    – Nhưng bạn không thể tra cứu tất cả các số giấy phép đó kịp thời," Drake phản đối. "Chúng tôi không cần phải làm vậy, Paul. Chúng tôi chỉ sắp xếp 1 danh sách & tìm kiếm sự trùng lặp.

  - ""Treesort" Computer – With this new 'computer-approach' to nature study you can quickly identify over 260 different trees of U.S., Alaska, & Canada, even palms, desert trees, & other exotics. To sort, you simply insert the needle." – Edmund Scientific Company, *Catalog* (1964)

    – Máy tính "Treesort" – Với 'cách tiếp cận máy tính' mới này đối với nghiên cứu thiên nhiên, bạn có thể nhanh chóng xác định hơn 260 loại cây khác nhau của Hoa Kỳ, Alaska, & Canada, thậm chí cả cây cọ, cây sa mạc, & các loại cây ngoại lai khác. Để phân loại, bạn chỉ cần chèn kim.

  In this chap, study a topic that arises frequently in programming: rearrangement of items into ascending or descending order. Imagine how hard it would be to us a dictionary if its words were not alphabetized! In a similar way, order in which items are stored in computer memory often has a profound influence on speed & simplicity of algorithms that manipulate those items.

  – Trong chương này, hãy nghiên cứu 1 chủ đề thường gặp trong lập trình: sắp xếp lại các mục theo thứ tự tăng dần hoặc giảm dần. Hãy tưởng tượng xem sẽ khó khăn như thế nào đối với chúng ta khi sử dụng từ điển nếu các từ trong đó không được sắp xếp theo thứ tự bảng chữ cái! Tương tự như vậy, thứ tự các mục được lưu trữ trong bộ nhớ máy tính thường có ảnh hưởng sâu sắc đến tốc độ & tính đơn giản của các thuật toán xử lý các mục đó.

  Although dictionaries of English language define "sorting" as process of separating or arranging things according to class or kind, computer programmers traditionally use word in much more special sense of marshaling things into ascending or descending order. Process should perhaps be called *ordering*, not sorting; but anyone who tries to call it "ordering" is soon led into confusion because of many different meanings attached to that word. Consider following sentence, e.g.: "Since only 2 of our tape drives were in working order, I was ordered to order more tape units in short order, in order to order data several orders of magnitude faster." Mathematical terminology abounds with still more senses of order (order of a group, order of a permutation, order of a branch point, relations of orders, etc., etc.). Thus find word "order" can lead to chaos.

  – Mặc dù các từ điển tiếng Anh định nghĩa "sắp xếp" là quá trình tách hoặc sắp xếp mọi thứ theo lớp hoặc loại, các lập trình viên máy tính theo truyền thống sử dụng từ này theo nghĩa đặc biệt hơn nhiều là sắp xếp mọi thứ theo thứ tự tăng dần hoặc giảm dần. Quá trình có lẽ nên được gọi là sắp xếp, không phải sắp xếp; nhưng bất kỳ ai cố gắng gọi nó là "sắp xếp" sẽ sớm bị nhầm lẫn vì có nhiều nghĩa khác nhau gắn liền với từ đó. Hãy xem xét câu sau, ví dụ: "Vì chỉ có 2 ổ băng của chúng tôi hoạt động bình thường, nên tôi được lệnh đặt hàng thêm các đơn vị băng trong thời gian ngắn, để sắp xếp dữ liệu nhanh hơn nhiều cấp độ." Thuật ngữ toán học có rất nhiều nghĩa về thứ tự (thứ tự của 1 nhóm, thứ tự của 1 hoán vị, thứ tự của 1 điểm nhánh, mối quan hệ của các thứ tự, v.v., v.v.). Do đó, tìm từ "trật tự" có thể dẫn đến hỗn loạn.

  Some people have suggested that "sequencing" would be best name for process of sorting into order; but this word often seems to lack right connotation, especially when equal elements are present, & it occasionally conflicts with other terminology. It is quite true that "sorting" is itself an overused word ("I was sort of out of sorts after sorting that sort of data"), but it has become firmly established in computing parlance. Therefore we shall use word "sorting" chiefly in strict sense of sorting into order, without further apologies.

  – 1 số người đã gợi ý rằng "sequencing" sẽ là tên gọi tốt nhất cho quá trình sắp xếp theo thứ tự; nhưng từ này thường có vẻ thiếu đúng nghĩa, đặc biệt là khi có các phần tử bằng nhau, & đôi khi nó xung đột với các thuật ngữ khác. Đúng là "sorting" tự nó là 1 từ bị lạm dụng ("Tôi đã hơi lạc lõng sau khi sắp xếp loại dữ liệu đó"), nhưng nó đã trở nên vững chắc trong thuật ngữ máy tính. Do đó, chúng ta sẽ sử dụng từ "sorting" chủ yếu theo nghĩa chặt chẽ là sắp xếp theo thứ tự, mà không cần xin lỗi thêm.

  Some of most important applications of sorting are:

  1. *Solving "togetherness" problem*, in which all items with same identification are brought together. Suppose: we have 10000 items in arbitrary order, many of which have equal values; & suppose we want to rearrange data so that all items with equal values appear in consecutive positions. This is essentially the problem of sorting in older sense of word; & it can be

solved easily by sorting file in new sense of word, so that values are in ascending order, $v_1 \le v_2 \le \cdots \le v_{10000}$. Efficiency achievable in this procedure explains why original meaning of "sorting" has changed.

– *Giải quyết vấn đề "cùng nhau"*, trong đó tất cả các mục có cùng định danh được đưa lại với nhau. Giả sử: chúng ta có 10000 mục theo thứ tự tùy ý, nhiều mục trong số đó có giá trị bằng nhau; & giả sử chúng ta muốn sắp xếp lại dữ liệu sao cho tất cả các mục có giá trị bằng nhau xuất hiện ở các vị trí liên tiếp. Về cơ bản, đây là vấn đề sắp xếp theo nghĩa cũ của từ; & có thể giải quyết dễ dàng bằng cách sắp xếp tệp theo nghĩa mới của từ, sao cho các giá trị theo thứ tự tăng dần, $v_1 \le v_2 \le \cdots \le v_{10000}$. Hiệu quả đạt được trong quy trình này giải thích tại sao nghĩa gốc của "sắp xếp" đã thay đổi.

2. *Matching items in ≥ 2 files.* If several files have been sorted into same order, possible to find all of matching entries in 1 sequential pass through them, without backing up. This is principle that Perry Mason used to help solve a murder case. We can usually process a list of information most quickly by traversing it in sequence from beginning to end, instead of skipping around at random in list, unless entire list is small enough to fit in a high-speed random-access memory. Sorting makes it possible to use sequential accessing on large files, as a feasible substitute for direct addressing.

   – *Các mục khớp trong các tệp ≥ 2.* Nếu 1 số tệp đã được sắp xếp theo cùng 1 thứ tự, có thể tìm thấy tất cả các mục khớp trong 1 lần duyệt tuần tự qua chúng, mà không cần sao lưu. Đây là nguyên tắc mà Perry Mason đã sử dụng để giúp giải quyết 1 vụ án mạng. Chúng ta thường có thể xử lý danh sách thông tin nhanh nhất bằng cách duyệt theo trình tự từ đầu đến cuối, thay vì bỏ qua ngẫu nhiên trong danh sách, trừ khi toàn bộ danh sách đủ nhỏ để vừa với bộ nhớ truy cập ngẫu nhiên tốc độ cao. Sắp xếp giúp có thể sử dụng truy cập tuần tự trên các tệp lớn, như 1 giải pháp thay thế khả thi cho việc định địa chỉ trực tiếp.

3. *Searching for information by key values.* Sorting is also an aid to searching, hence it helps us make computer output more suitable for human consumption. In fact, a listing that has been sorted into alphabetic order often looks quite authoritative even when associated numerical information has been incorrectly computed.

   – *Tìm kiếm thông tin theo giá trị khóa.* Sắp xếp cũng là 1 công cụ hỗ trợ tìm kiếm, do đó giúp chúng ta làm cho đầu ra máy tính phù hợp hơn với nhu cầu sử dụng của con người. Trên thực tế, 1 danh sách được sắp xếp theo thứ tự chữ cái thường trông khá có thẩm quyền ngay cả khi thông tin số liên quan đã được tính toán không chính xác.

Although sorting has traditionally been used mostly for business data processing, it is actually a basic tool that every programmer should keep in mind for use in a wide variety of situations. We have discussed its use for simplifying algebraic formulas, in exercise 2.3.2–17. Exercises below illustrate diversity of typical applications.

– Mặc dù sắp xếp theo truyền thống chủ yếu được sử dụng để xử lý dữ liệu kinh doanh, nhưng thực tế đây là 1 công cụ cơ bản mà mọi lập trình viên nên ghi nhớ để sử dụng trong nhiều tình huống khác nhau. Chúng tôi đã thảo luận về việc sử dụng nó để đơn giản hóa các công thức đại số trong bài tập 2.3.2–17. Các bài tập dưới đây minh họa sự đa dạng của các ứng dụng điển hình.

1 of 1st large-scale software systems to demonstrate versatility of sorting was LARC Scientific Compiler developed by J. Erdwinn, D. E. Ferguson, & their associates at Computer Sciences Corporation in 1960. This optimizing compiler for an extended FORTRAN language made heavy use of sorting so that various compilation algorithms were presented with relevant parts of source program in a convenient sequence. 1st pass was a lexical scan that divided FORTRAN source code into individual tokens, each representing an identifier or a constant or an operator, etc. Each token was assigned several sequence numbers; when sorted on name & an appropriate sequence number, all uses of a given identifier were brought together. "Defining entries" by which a user would specify whether an identifier stood for a function name, a parameter, or a dimensioned variable were given low sequence numbers, so that they would appear 1st among tokens having a given identifier; this made it easy to check for conflicting usage & to allocate storage w.r.t. `EQUIVALENCE` declarations. Information thus gathered about each identifier was now attached to each token; in this way no "symbol table" of identifiers needed to be maintained in high-speed memory. Updated tokens were then sorted on another sequence number, which essentially brought source program back into its original order except that numbering scheme was cleverly designed to put arithmetic expressions into a more convenient "Polish prefix" form. Sorting was also used in later phases of compilation, to facilitate loop optimization, to merge error messages into listing, etc. In short, compiler was designed so that virtually all processing could be done sequentially from files that were stored in an auxiliary drum memory, since appropriate sequence numbers were attached to data in such a way that it could be sorted into various convenient arrangements.

– 1 trong những hệ thống phần mềm quy mô lớn đầu tiên chứng minh tính linh hoạt của việc sắp xếp là LARC Scientific Compiler do J. Erdwinn, D. E. Ferguson, & các cộng sự của họ tại Computer Sciences Corporation phát triển vào năm 1960. Trình biên dịch tối ưu hóa này cho ngôn ngữ FORTRAN mở rộng đã sử dụng rất nhiều sắp xếp để các thuật toán biên dịch khác nhau được trình bày với các phần có liên quan của chương trình nguồn theo trình tự thuận tiện. Lần quét đầu tiên là quét từ vựng chia mã nguồn FORTRAN thành các mã thông báo riêng lẻ, mỗi mã thông báo đại diện cho 1 mã định danh hoặc 1 hằng số hoặc 1 toán tử, v.v. Mỗi mã thông báo được gán 1 số số thứ tự; khi được sắp xếp theo tên & 1 số thứ tự thích hợp, tất cả các lần sử dụng của 1 mã định danh nhất định sẽ được đưa lại với nhau. "Các mục nhập định nghĩa" mà người dùng sẽ chỉ định xem 1 mã định danh có đại diện cho tên hàm, tham số hay biến có kích thước được gán các số thứ tự thấp để chúng xuất hiện đầu tiên trong số các mã thông báo có mã định danh nhất định; điều này giúp dễ dàng kiểm tra việc sử dụng xung đột & để phân bổ dung lượng lưu trữ cho các khai báo `EQUIVALENCE`. Thông tin thu thập được về mỗi mã định danh như vậy giờ đây được đính kèm vào mỗi mã thông báo; theo cách này, không cần phải duy trì "bảng ký hiệu" của các mã định danh trong bộ nhớ tốc độ cao. Các mã thông báo được cập nhật sau đó được sắp xếp theo 1 số thứ tự khác, về cơ bản là đưa chương trình nguồn trở lại thứ tự ban đầu của nó ngoại trừ lược đồ đánh số được thiết kế khéo léo để đưa các biểu thức số học vào dạng "tiền tố Ba Lan" thuận tiện hơn. Sắp xếp cũng được sử dụng trong các giai đoạn biên dịch sau này, để

tạo điều kiện tối ưu hóa vòng lặp, để hợp nhất các thông báo lỗi vào danh sách, v.v. Tóm lại, trình biên dịch được thiết kế sao cho hầu như mọi quá trình xử lý đều có thể được thực hiện tuần tự từ các tệp được lưu trữ trong bộ nhớ trống phụ, vì các số thứ tự thích hợp được đính kèm vào dữ liệu theo cách có thể sắp xếp dữ liệu thành nhiều cách sắp xếp thuận tiện khác nhau.

Computer manufacturers of 1960s estimated $> 25\%$ of running time on their computers was spent on sorting, when all customers were taken into account. In fact, there were many installations in which task of sorting was responsible for $> \frac{1}{2}$ of computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. Real truth probably involves all 3 of these possibilities, but in any event we can see: sorting is worthy of serious study, as a practical matter.

– Các nhà sản xuất máy tính của những năm 1960 ước tính $> 25\%$ thời gian chạy trên máy tính của họ được dành cho việc sắp xếp, khi tất cả khách hàng đều được tính đến. Trên thực tế, có nhiều cài đặt trong đó nhiệm vụ sắp xếp chịu trách nhiệm cho $> \frac{1}{2}$ thời gian tính toán. Từ các số liệu thống kê này, chúng ta có thể kết luận rằng (i) có nhiều ứng dụng quan trọng của việc sắp xếp, hoặc (ii) nhiều người sắp xếp khi họ không nên, hoặc (iii) các thuật toán sắp xếp không hiệu quả đã được sử dụng phổ biến. Sự thật thực sự có thể bao gồm cả 3 khả năng này, nhưng trong mọi trường hợp, chúng ta có thể thấy: sắp xếp đáng được nghiên cứu nghiêm túc, như 1 vấn đề thực tế.

Even if sorting were almost useless, there would be plenty of rewarding reasons for studying it anyway! Ingenious algorithms that have been discovered show: sorting is an extremely interesting topic to explore in its own right. Many fascinating unsolved problems remain in this area, as well as quite a few solved ones.

– Ngay cả khi sắp xếp gần như vô dụng, vẫn có rất nhiều lý do đáng để nghiên cứu nó! Các thuật toán khéo léo đã được phát hiện cho thấy: sắp xếp là 1 chủ đề cực kỳ thú vị để khám phá theo cách riêng của nó. Nhiều vấn đề chưa được giải quyết hấp dẫn vẫn còn trong lĩnh vực này, cũng như khá nhiều vấn đề đã được giải quyết.

From a broader perspectives we will find also: sorting algorithms make a valuable *case study* of how to attack computer programming problems in general. Many important principles of data structure manipulation will be illustrated in this chap. We will be examining evolution of various sorting techniques in an attempt to indicate how ideas were discovered in 1st place. By extrapolating this case study, can learn a good deal about strategies that help us design good algorithms for other computer problems.

– Từ góc nhìn rộng hơn, chúng ta cũng sẽ thấy: các thuật toán sắp xếp tạo ra 1 *case study* có giá trị về cách giải quyết các vấn đề lập trình máy tính nói chung. Nhiều nguyên tắc quan trọng về thao tác cấu trúc dữ liệu sẽ được minh họa trong chương này. Chúng ta sẽ xem xét sự tiến hóa của nhiều kỹ thuật sắp xếp khác nhau nhằm chỉ ra cách các ý tưởng được phát hiện ngay từ đầu. Bằng cách suy rộng nghiên cứu trường hợp này, có thể học được nhiều điều về các chiến lược giúp chúng ta thiết kế các thuật toán tốt cho các vấn đề máy tính khác.

Sorting techniques also provide excellent illustrations of general ideas involved in *analysis of algorithms* – ideas used to determine performance characteristics of algorithms so that an intelligent choice can be made between competing methods. Readers who are mathematically inclined will find quite a few instructive techniques in this chap for estimating speed of computer algorithms & for solving complicated recurrence relations. On other hand, material has been arranged so that readers without a mathematical bent can safely skip over these calculations.

– Các kỹ thuật sắp xếp cũng cung cấp những minh họa tuyệt vời về các ý tưởng chung liên quan đến *phân tích thuật toán* – các ý tưởng được sử dụng để xác định các đặc điểm hiệu suất của thuật toán để có thể đưa ra lựa chọn thông minh giữa các phương pháp cạnh tranh. Những độc giả có khuynh hướng toán học sẽ tìm thấy khá nhiều kỹ thuật hướng dẫn trong chương này để ước tính tốc độ của các thuật toán máy tính & để giải các mối quan hệ đệ quy phức tạp. Mặt khác, tài liệu đã được sắp xếp để những độc giả không có khuynh hướng toán học có thể bỏ qua các phép tính này 1 cách an toàn.

p. 5+++

- 5.1. Combinatorial Properties of Permutations.
- 5.2. Internal Sorting.
- 5.3. Optimum Sorting.
- 5.4. External Sorting.
- 5.5. Summary, History, & Bibliography.

- Chap. 6: Searching.
  - 6.1. Sequential Searching.
  - 6.2. Searching by Comparison of Keys.
  - 6.3. Digital Searching.
  - 6.4. Hashing.
  - 6.5. Retrieval on Secondary Keys.

- Appendix A: Tables of Numerical Quantities.
  - 1. Fundamental Constants (decimal).
  - 2. Fundamental Constants (octal).

- 3. Harmonic Numbers, Bernoulli Numbers, Fibonacci Numbers.

- Appendix B: Index to Notations.
- Appendix C: Index to Algorithms & Theorems.

2.3 [Knu11]. Donald Erwin Knuth. The Art of Computer Programming. Vol. 4: Combinatorial Algorithms. Part 1. 1e

2.4 [Knu11]. Donald Erwin Knuth. The Art of Computer Programming. Vol. 4: Combinatorial Algorithms. Part 2. 1e

3 Wikipedia's

4 Miscellaneous

Tài liệu

[Cor+22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 4th edition. MIT Press, 2022, p. 1312.

[Knu11] Donald Ervin Knuth. *The Art of Computer Programming. Volume 4A: Combinatorial Algorithms. Part 1*. 2nd edition. Addison-Wesley Professional, 2011, p. 912.

[Knu98] Donald Ervin Knuth. *The Art of Computer Programming. Volume 3: Sorting & Searching*. 2nd edition. Addison-Wesley Professional, 1998, p. 800.