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

Nguyễn Quản Bá Hồng*

Ngày 25 tháng 5 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 **1**

2 Algorithms – Thuật Giải/Thuật Toán **1**
2.1 [Cor+22]. Introduction to Algorithms. 4e 1

3 Wikipedia's **9**

4 Miscellaneous **9**

Tài liệu **9**

1 Basic Data Structure – Cấu Trúc Dữ Liệu Cơ Bản

2 Algorithms – Thuật Giải/Thuật Toán

2.1 [Cor+22]. 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:
  - 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

*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.

- 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.
  - 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.
    p. 28
- 2. Getting Started.
- 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 một 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 một đồ thị có hướng không có chu trình & phân tích một 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, & một 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 một 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 một thuật toán đồ thị trên một 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 một 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 một đồ thị & để tìm kiếm một đồ thị. Tìm kiếm một đồ thị có nghĩa là theo dõi một 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 một 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 một 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 một 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 một 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 một 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.

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.