Computer – Máy Tính

Nguyễn Quản Bá Hồng*

Ngày 7 tháng 4 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:

- *Computer – Máy Tính.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.tex.

Mục lục

1 Algorithms . . . **2**
1.1 [Cor+22]. Introduction to Algorithms. 4e . . . 2

2 Donald Knuth. The Art of Computer Programming. Volume 1: Fundamental Algorithms . . . **9**
2.1 Preface . . . 9
2.2 Chap. 1: Basic Concepts . . . 14
2.2.1 Algorithms . . . 14
2.3 Chap. 2: Information Structures . . . 14

3 Competitive Programming (CP) . . . **14**
3.1 [DV21]. Christoph Dürr, Jill-Jênn Vie. Competitive Programming in Python: 128 Algorithms to Develop Your Coding Skills. 2021 . . . 14
3.2 [Laa20; Laa24]. Antti Laaksonen. Guide to Competitive Programming: Learning & Improving Algorithms Through Contests . . . 19
3.3 [WW16]. Yonghui Wu, Jiande Wang. Data Structure Practice for Collegiate Programming Contests & Education . . . 29
3.4 [WW18]. Yonghui Wu, Jiande Wang. Algorithm Design Practice for Collegiate Programming Contests & Education . . . 33

4 Conda . . . **34**
4.1 Should I use Anaconda Distribution or Miniconda? . . . 34
4.2 Anaconda Distribution . . . 35
4.2.1 System requirements . . . 35
4.2.2 Installing Anaconda Distribution . . . 35
4.2.3 Advanced installation . . . 37
4.2.4 Getting started . . . 37
4.2.5 Anaconda Distribution release notes . . . 38
4.2.6 Uninstalling Anaconda Distribution . . . 38
4.3 Miniconda . . . 38
4.3.1 Latest Miniconda installer links . . . 38
4.3.2 Quick command line install . . . 38

5 LaTeX . . . **38**
5.1 Overleaf/glossaries . . . 38

6 Linux . . . **38**

7 Programming . . . **38**
7.1 C/C++ . . . 38
7.2 Pascal . . . 38
7.3 Python . . . 39

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

8 Software . . . **39**
8.1 FeNiCS . . . 39
8.2 Firedrake . . . 39
8.3 Fireshape . . . 39
8.4 Git . . . 39
8.5 Gmsh . . . 39
8.6 OpenFOAM . . . 39
8.7 ParMooN . . . 40
8.8 SU2 . . . 40
8.9 Sublime Text . . . 40

9 SymPy . . . **40**
9.1 Matrices . . . 40

10 Miscellaneous . . . **47**

11 Wikipedia . . . **47**
11.1 Wikipedia/abstraction (computer science) . . . 47
11.1.1 Rationale . . . 47
11.1.2 Abstraction features . . . 48
11.1.3 Control abstraction . . . 48
11.1.4 Data abstraction . . . 49
11.1.5 Manual data abstraction . . . 49
11.1.6 Abstraction in OOP . . . 49
11.1.7 Considerations . . . 50
11.1.8 Levels of abstraction . . . 51
11.2 Wikipedia/Anaconda . . . 51
11.2.1 History . . . 52
11.2.2 Overview . . . 52
11.2.3 Anaconda.org . . . 53
11.3 Wikipedia/computational learning theory . . . 53
11.3.1 Overview . . . 53
11.4 Wikipedia/Conda (package manager) . . . 53
11.4.1 Features . . . 54
11.5 Wikipedia/data structure . . . 54
11.5.1 Usage . . . 54
11.5.2 Implementation . . . 54
11.5.3 Examples . . . 54
11.5.4 Language support . . . 55
11.6 Wikipedia/level set (data structures) . . . 55
11.6.1 Chronological developments . . . 55
11.7 Wikipedia/pip (package manager) . . . 56
11.7.1 History . . . 56
11.7.2 Command-line interface . . . 56
11.7.3 Custom repository . . . 57

Tài liệu . . . **57**

1 Algorithms

1.1 [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
- 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.

- 20. Elementary Graph Algorithms.
- 21. Minimum Spanning Trees.
- 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 Donald Knuth. The Art of Computer Programming. Volume 1: Fundamental Algorithms

Resources – Tài nguyên.

1. [Knu97]. Donald Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms.*

   Với bản dịch tiếng Việt chứa nhiều lỗi chính tả đến mức đáng bị cảnh sát ghé thăm:

2. [Kho06]. Nguyễn Văn Khoa. *Nghệ Thuật Lập Trình Máy Tính.*

This series of books is affectionately dedicated to the Type 650 computer once installed at Case Institute of Technology, in remembrance of many pleasant evenings.

The author & publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind & assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

2.1 Preface

"Here is your book, the one your thousands of letters have asked us to publish. It has taken us years to do, checking & rechecking countless recipes to bring you only the best, only the interesting, only the perfect. Now we can say, without a shadow of a doubt, that every single 1 of them, if you follow the directions to the letter, will work for you exactly as well as it did for us, even if you have never cooked before." – McCall's Cookbook (1963)

"The process of preparing programs for a digital computer is especially attractive, not only because it can be economically & scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. This book is the 1st volume of a multi-volume set of books that has been designed to train the reader in various skills that go into a programmer's craft.

The following chapters are *not* meant to serve as an introduction to computer programming; the reader is supposed to have had some previous experience. The prerequisites are actually very simple, but a beginner requires time & practice in order to understand the concept of a digital computer. The reader should possess:

1. Some idea of how a stored-program digital computer works; not necessarily the electronics, rather the manner in which instructions can be kept in the machine's memory & successively executed.

2. An ability to put the solutions to problems into such explicit terms that a computer can "understand" them. (These machines have no common sense; they do exactly as they are told, no more & no less. This fact is the hardest concept to grasp when one 1st tries to use a computer.)

3. Some knowledge of the most elementary computer techniques, e.g. looping (performing a set of instructions repeatedly), the use of subroutines, & the use of indexed variables.

4. A little knowledge of common computer jargon – "memory," "registers," "bits," "floating point," "overflow," "software." Most words not defined in the text are given brief definitions in the index at the close of each volume.

These 4 prerequisites can perhaps be summed up into the single requirement that the reader should have written & tested at least, say, 4 programs for at least 1 computer.

I have tried to write this set of books in such a way that it will fill several needs. In the 1st place, these books are reference works that summarize the knowledge that has been acquired in several important fields. In the 2nd place, they can be used as textbooks for self-study or for college courses in the computer & information sciences. To meet both of these objectives, I have incorporated a large number of exercises into the text & have furnished answers for most of them. I have also made an effort to fill the pages with facts rather than with vague, general commentary.

This set of books is intended for people who will be more than just casually interested in computers, yet it is by no means only for the computer specialist. Indeed, 1 of my main goals has been to make these programming techniques more accessible to the many people working in other fields who can make fruitful use of computers, yet who cannot afford the time to locate all of the necessary information that is buried in technical journals.

We might call the subject of these books "nonumerical analysis." Computers have traditionally been associated with the solution of numerical problems e.g. calculation of roots of an equation, numerical interpolation & integration, etc., but such topics are not treated here except in passing. Numerical computer programming is an extremely interesting & rapidly expanding field, & many books have been written about it. Since the early 1960s, however, computers have been used even more often for problems in which numbers occur only by coincidence; the computer's decision-making capabilities are being used, rather than its ability to do arithmetic. We have some use for addition & subtraction in nonnumerical problems, but we rarely feel any need for multiplication & division. Of course, even a person who is primarily concerned with numerical computer programming will benefit from a study of the nonnumerical techniques, for they are present in the background of numerical programs as well.

The results of research in nonnumerical analysis are scattered throughout numerous technical journals. My approach has been to try to distill this vast literature by studying the techniques that are most basic, in the sense that they can be applied to many types of programming situations. I have attempted to coordinate the ideas into more or less of a "theory," as well as to show how the theory applies to a wide variety of practical problems.

Of course, "nonnumerical analysis" is a terribly negative name for this field of study; it is much better to have a positive, descriptive term that characterizes the subject. "Information processing" is too broad a designation for the material I am considering, & "programming techniques" is too narrow. Therefore I wish to propose *analysis of algorithms* as an appropriate name for the subject matter covered in these books. This name is meant to imply "the theory of the properties of particular computer algorithms."

The complete set of books, entitled *The Art of Computer Programming*, has the following general outline:

1. *Vol. 1: Fundamental Algorithms.* Chap. 1: Basic Concepts. Chap. 2: Information Structures.

2. *Vol. 2: Seminumerical Algorithms.* Chap. 3: Random Numbers. Chap. 4: Arithmetic.

3. *Vol. 3: Sorting & Searching.* Chap. 5: Sorting. Chap. 6: Searching.

4. *Vol. 4: Combinatorial Algorithms.* Chap. 7: Combinatorial Searching. Chap. 8: Recursion.

5. *Vol. 5: Syntactical Algorithms.* Chap. 9: Lexical Scanning. Chap. 10: Parsing.

Vol. 4 deals with such a large topic, it actually represents several separate books (Vols. 4A, 4B, etc.). 2 additional volumes on more specialized topics are also planned: Vol. 6: *The theory of Languages* (Chap. 11)' Vol. 7: *Compilers* (Chap. 12).

I started out in 1962 to write a single book with this sequence of chapters, but I soon found that it was more important to treat the subjects in depth rather than to skim over them lightly. The resulting length of the text has meant that each chapter by itself contains more than enough material for a 1-semester college course; so it has become sensible to publish the series in separate volumes. I know that it is strange to have only 1 or 2 chapters in an entire book, but I have decided to retain the original chapter numbering in order to facilitate cross references. A shorter version of Vols. 1–5 is planned, intended specifically to serve as a more general reference &/or text for undergraduate computer courses; its contents will be a subset of the material in these books, with the more specialized information omitted. The same chapter numbering will be used in the abridged edition as in the complete work.

The present volume may be considered as the "intersection" of the entire set, in the sense that it contains basic material that is used in all the other books. Vols. 2–5, on the other hand, may be read independently of each other. Vol. 1 is not only a reference book to be used in connection with the remaining volumes; it may also be used in college courses or for self-study as a

text on the subject of *data structures* (emphasizing the material of Chap. 2), or as a text on the subject of *discrete mathematics*, or as a text on the subject of *machine-language programming*.

The point of view I have adopted while writing these chapters differs from that taken in most contemporary books about computer programming in that I am not trying to teach the reader how to use somebody else's software. I am concerned rather with teaching people how to write better software themselves.

My original goal was to bring readers to the frontiers of knowledge in every subject that was treated. But it is extremely difficult to keep up with a field that is economically profitable, & the rapid rise of computer science has made such a dream impossible. The subject has become a vast tapestry[1] with $a \cdot 10^4$ of subtle results contributed by $a \cdot 10^4$ of talented people all over the world. Therefore my new goal has been to concentrate on "classic" techniques that are likely to remain important for many more decades, & to describe them as well as I can. In particular, I have tried to trace the history of each subject, & to provide a solid foundation for future progress. I have attempted to choose terminology that is concise & consistent with current usage. I have tried to include all of the known ideas about sequential computer programming that are both beautiful & easy to state.

A few words are in order about the mathematical content of this set of books. The material has been organized so that persons with no more than a knowledge of high-school algebra may read it, skimming briefly over the more mathematical portions; yet a reader who is mathematically inclined will learn about many interesting mathematical techniques related to discrete mathematics. This dual level of presentation has been achieved in part by assigning ratings to each of the exercises so that the primarily mathematical ones are marked specifically as such, & also by arranging most sections so that the main mathematical results are stated *before* their proofs. The proofs are either left as exercises (with answers to be found in a separate section) or they are given at the end of a section.

A reader who is interested primarily in programming rather than in the associated mathematics may stop reading most sections as soon as the mathematics becomes recognizably difficult. On the other hand, a mathematically oriented reader will find a wealth of interesting material collected here. Much of the published mathematics about computer programming has been faulty, & 1 of the purposes of this book is to instruct readers in proper mathematical approaches to this subject. Since I profess to be a mathematician, it is my duty to maintain mathematical integrity as well as I can.

A knowledge of elementary calculus will suffice for most of the mathematics in these books, since most of the other theory that is needed is developed herein. However, I do need to use deeper theorems of complex variable theory, probability theory, number theory, etc., at times, & in such cases I refer to appropriate textbooks where those subjects are developed.

The hardest decision that I had to make while preparing these books concerned the manner in which to present the various techniques. The advantages of flow charts & of an informal step-by-step description of an algorithm are well known; for a discussion of this, see the article "Computer-Drawn Flowcharts" in the ACM *Communications*, Vol. 6 (Sep. 1963), pp. 555–563. Yet a formal, precise language is also necessary to specify any computer algorithm, & I needed to decide whether to use an algebraic language, e.g. ALGOL or FORTRAN, or to use a machine-oriented language for this purpose. Perhaps many of today's computer experts will disagree with my decision to use a machine-oriented language, but I have become convinced that it was definitely the correct choice, for the following reasons:

1. A programmer is greatly influenced by the language in which programs are written; there is an overwhelming tendency to prefer constructions that are simplest in that language, rather than those that are best for the machine. By understanding a machine-oriented language, the programmer will tend to use a much more efficient method; it is much closer to reality.

2. The programs we require are, with a few exceptions, all rather short, so with a suitable computer there will be no trouble understanding the programs.

3. High-level languages are inadequate for discussing important low-level details e.g. coroutine linkage, random number generation, multi-precision arithmetic, & many problems involving the efficient usage of memory.

4. A person who is more than casually interested in computers should be well schooled in machine language, since it is a fundamental part of a computer.

5. Some machine language would be necessary anyway as output of the software programs described in many of the examples.

6. New algebraic languages go in & out of fashion every 5 years or so, while I am trying to emphasize concepts that are timeless.

From the other point of view, I admit that it is somewhat easier to write programs in higher-level programming languages, & it is considerably easier to debug the programs. Indeed, I have rarely used low-level machine language for my own programs since 1970, now that computers are so large & so fast. Many of the problems of interest to us in this book, however, are those for which the programmer's art is most important. E.g., some combinatorial calculations need to be repeated a trillion times, & we save about 11.6 days of computation for every microsecond we can squeeze out of their inner loop. Similarly, it is worthwhile to put an additional effort into the writing of software that will be used many times each day in many computer installations, since the software needs to be written only once.

Given the decision to use a machine-oriented language, which language should be used? I could have chosen the language of a particular machine X, but then those people who do not possess machine X would think this book is only for X-people. Furthermore, machine X probably has a lot of idiosyncrasies[2] that are completely irrelevant to the material in this book yet

[1] tấm thảm.
[2] đặc điểm riêng.

which must be explained; & in 2 years the manufacturer of machine X will put out machine $X+1$ or machine $10X$, & machine X will no longer be of interest to anyone.

To avoid this dilemma, I have attempted to design an "ideal" computer with very simple rules of operation (requiring, say, only an hour to learn), which also resembles actual machines very closely. There is no reason why a student should be afraid of learning the characteristics of > 1 computer; once 1 machine language has been mastered, others are easily assimilated. Indeed, serious programmers may expect to meet different machine languages in the course of their careers. So the only remaining disadvantage of a mythical machine is the difficulty of executing any programs written for it. Fortunately, that is not really a problem, because many volunteers have come toward to write simulators for the hypothetical machine. Such simulators are ideal for instructional purposes, since they are even easier to use than a real computer would be. I have attempted to cite the best early papers in each subject, together with a sampling of more recent work. When referring to the literature, I use standard abbreviations for the names of periodicals, except that the most commonly cited journals are abbreviated as follows:

1. CACM = Communications of the Association for Computing Machinery
2. CACMJACM = Journal of the Association for Computing Machinery
3. CACMComp. J. = The Computer Journal (British Computer Society)
4. CACMMath. Comp. = Mathematics of Computation
5. CACMAMM = American Mathematical Monthly
6. CACMSICOMP = SIAM Journal on Computing
7. CACMFOCS = IEEE Symposium on Foundations of Computer Science
8. CACMSODA = ACM–SIAM Symposium on Discrete Algorithms
9. CACMSTOC = ACM Symposium on Theory of Computing
10. CACMCrelle = Journal für die reine und angewandte Mathematik

I also use "CMath" to stand for the book *Concrete Mathematics*.

Preface to the 3rd Edition. After having spent 10 years developing the TEX & METAFONT systems for computer typesetting, I am now able to fulfill the dream that I had when I began that work, by applying those systems to *The Art of Computer Programming*. At last the entire text of this book has been captured inside my personal computer, in an electronic form that will make it readily adaptable to future changes in printing & display technology. The new setup has allowed me to make literally thousands of improvements that I've been wanting to incorporate for a long time.

In this new edition I have gone over every word of the text, trying to retain the youthful exuberance[3] of my 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.

The Art of Computer Programming is, however, still a work in progress. Therefore some parts of this book are headed by an "under construction" icon, to apologize for the fact that the material is not up-to-date. My files are bursting with important material that I plan to include in the final, glorious, 4th edition of Vol. 1, perhaps 15 years from now; but I must finish Vols. 4 –5 1st, & I do not want to delay their publication any more than absolute necessary.

My efforts to extend & enhance these volumes have been enormously enhanced since 1980 by the wise guidance of Addison–Wesley's editor Peter Gordon. He has become not only my "publishing partner" but also a close friend, while continually nudging me to move in fruitful directions. Indeed, my interactions with dozens of Addison–Wesley people during > 3 decades have been much better than any author deserves. The tireless support of managing editor John Fuller, whose meticulous attention to detail has maintained the highest standards of production quality in spite of frequent updates, has been particularly praiseworthy.

> *"Things have changed in the past 2 decades."* – Bill Gates (1995)
>
> *"Woe be to him that reads but one book."* – George Herbert, *Jacula Prudentum*, 1144 (1640)
>
> *"Le défaut unique de tous les ouvrages c'est d'être trop longs."* – Vauvenargues, *Réflexions*, 628 (1746)
>
> *"Books are a triviality*[4]*. Life alone is great."* – Thomas Carlyle, *Journal* (1839)

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

In many books, easy exercises are found mixed randomly among extremely difficult ones. A motley[5] mixture is, however, often unfortunate because readers like to know in advance how long a problem ought to take – otherwise they may just skip

[3] sự hồ hởi.
[4] sự tầm thường.
[5] sặc sỡ.

over all the problems. A classic example of such a situation is the book *Dynamic Programming* by RICHARD BELLMAN; this is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading "Exercises & Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. BELLMAN how to tell the exercises apart from the research problems, & he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems & very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance: Rating: Interpretation

- 00: An extremely easy exercise that can be answered immediately if the material of the text has been understood; such an exercise can almost always be worked "in your head."
- 10: A simple problem that makes you think over the material just read, but is by no means difficult. You should be able to do this in one minute at most; pencil & paper may be useful in obtaining the solution.
- 20: An average problem that tests basic understanding of the text material, but you may need about 15 or 20 minutes to answer it completely.
- 30: A problem of moderate difficulty &/or complexity; this one may involve $>$ 2 hours' work to solve satisfactorily, or even more if the TV is on.
- 40: Quite a difficult or lengthy problem that would be suitable for a term project in classroom situations. A student should be able to solve the problem in a reasonable amount of time, but the solution is not trivial.
- 50: A research problem that has not yet been solved satisfactorily, as far as the author knew at the time of writing, although many people have tried. If you have found an answer to such a problem, you ought to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided that it is correct).

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. E.g., a rating of 17 would indicate an exercise that is a bit simpler than average. Problems with a rating of 50 that are subsequently solved by some reader may appear with a 40 rating in later editions of the book, & in the errata posted on the Internet.

The remainder of the rating number divided by 5 indicates the amount of detailed work required. Thus, an exercise rated 24 may take longer to solve than an exercise that is rated 25, but the latter will require more creativity. All exercises with ratings of 46 or more are open problems for future research, rated according to the number of different attacks that they've resisted so far.

The author has tried earnestly to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else to find a solution; & everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess at the level of difficulty, but the should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training & sophistication; as a result, some of the exercises are intended only for the use of more mathematically inclined readers. The rating is preceded by an M if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested only in programming the algorithms themselves. An exercise is marked with the letters "HM" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "HM" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, ▷; this designates problems that are especially instructive & especially recommended. Of course, no reader/student is expected to work *all* of the exercises, so those that seem to be the most valuable have been singled out. (This distinction is not meant to detract from the other exercises!) Each reader should at least make an attempt to solve all of the problems whose rating is 10 or less; & the arrows may help to indicate which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you absolutely do not have time to work this particular problem. *After* getting your own solution or giving the problem a decent try, you may find the answer instructive & helpful. The solution given will often be quite short, & it will sketch the details under the assumption that you have earnestly tried to solve it by your own means 1st. Sometimes the solution gives less information than was asked; often it gives more. It is quite possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details. Later printings of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answers to previous exercises, unless specifically forbidden from doing so. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n, even though it includes the result of exercise n as a special case.

Summary of codes. ▷: Recommended. M: Mathematical oriented. HM: Requiring "higher math". 00: Intermediate. 10: Simple (1 min). 20: Medium (15 mins). 30: Moderately hard. 40: Term project. 50: Research problem.

2. [10] Of what value can the exercises in a textbook be to the reader? **3.** Generalize your answer. [This is an example of a horrible kind of problem that the author has tried to avoid.]

> *"We can face our problem. We can arrange such facts as we have with order & method."* – HERCULE POIROT, in *Murder on the Orient Express* (1934)

2.2 Chap. 1: Basic Concepts

> *"Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical & numerical, rather than algebraical & analytical. This is an error. The engine can arrange & combine its numerical quantities exactly as if they were letters or any other general symbols; & in fact it might bring out its results in algebraical notation, were provisions made accordingly."* – Augusta Ada, *Countess of Lovelace* (1843)
>
> *"Practice yourself, for heaven's sake, in little things; & thence proceed to greater."* – Epictetus, *Discourses* IV.i

2.2.1 Algorithms

The notion of an *algorithm* is basic to all of computer programming, so we should begin with a careful analysis of this concept.

The word "algorithm" itself is quite interesting; at 1st glance it may look as though someone intended to write "logarithm" but jumbled up the 1st 4 letters. The word did not appear in *Webster's New World Dictionary* as late as 1957; we find only the older form "algorism" with its ancient meaning, the process of doing arithmetic using Arabic numerals. During the Middle Ages, abacists computed on the abacus & algorists computed by algorism. By the time of the Renaissance, the origin of this word was in doubt, & early linguists attempted to guess at its derivation by making combinations like *algiros* [painful] + *arithmos* [number]; others said no, the word comes from "King Algor of Castile." Finally, historians of mathematics found the true origin of the word algorism: It comes from the name of a famous Persian textbook author, Abu 'Abd Allah Muh ammad ibn Musa al-Khwarizmi (c. 825) – literally, "Father of Abdullah, Mohammed, son of Moses, native of Khwarizm." The Aral Sea in Central Asia was once known as Lake Khwarizm, & the Khwarizm region is located in the Amu River basin just south of that sea. Al-Khwarizmi wrote the celebrated Arabic text *Kitab al-jabr wa'l-muqabala* ("Rules of restoring & equating"); another word, "algebra," stems from the title of that book, which was a systematic study of the solution of linear & quadratic equations.

Gradually the form & meaning of *algorism* became corrupted; as explained by the *Oxford English Dictionary*, the word "passed through many pseudo-etymological perversions, including a recent *algorithm*, in which it is learnedly confused" with the Greek root of the word *arithmetic*. This change from "algorism" to "algorithm" is not hard to understand in view of the fact that people had forgotten the original derivation of the word. An early German mathematical dictionary, *Vollständiges mathematisches Lexicon* (Leipzig: 1747), gave the following definition for the word *Algorithmus*: "Under this designation are combined the notions of the 4 types of arithmetic calculations, namely addition, multiplication, subtraction, & division." The Latin phrase *algorithmus infinitesimalis* was at that time used to denote "ways of calculation with infinitely small quantities, as invented by Leibniz."

By 1950, the word algorithm was most frequently associated with Euclid's algorithm, a process for finding the greatest common divisor of 2 numbers appearing in Euclid's *Elements* (Book 7, Props. 1–2), see [Knu97, Algorithm E: Euclid's algorithm, p. 2].

Each algorithm we consider has been given an identifying letter (E in the preceding example), & the steps of the algorithm are identified by this letter followed by a number (E1, E2, E3).

Each step of an algorithm, e.g. step E1 above, begins with a phrase in brackets that sums up as briefly as possible the principal content of that step. This phrase also usually appears in an accompanying *flow chart* so that the reader will be able to picture the algorithm more readily.

After the summarizing phrase comes a description in words & symbols of some *action* to be performed or some decision to be made. Parenthesized *comments*, like the 2nd sentence in step E1, may also appear. Comments are included as explanatory information about that step, often indicating certain invariant characteristics of the variables or the current goals. They do not specify actions belonging to the algorithm, but are meant only for the reader's benefit as possible aids to comprehension.

2.3 Chap. 2: Information Structures

3 Competitive Programming (CP)

3.1 [DV21]. Christoph Dürr, Jill-Jênn Vie. Competitive Programming in Python: 128 Algorithms to Develop Your Coding Skills. 2021

Want to kill it at your job interview in tech industry? Want to win that coding competition? Learn all algorithmic techniques & programming skills you need from 2 experienced coaches, problem-setters, & judges for coding competitions. Authors highlight versatility of each algorithms by considering a variety of problems & show how to implement algorithms in simple & efficient code. What to expect:

- Master 128 algorithms in Python.
- Discover right way to tackle a problem & quickly implement a solution of low complexity
- Understand classic problems like Dijkstra's shortest path algorithm & Knuth–Morris–Pratt's string matching algorithm, plus lesser-known data structures like Fenwick trees & Knuth's dancing links.
- Develop a framework to tackle algorithmic problem solving, including: Def, Complexity, Applications, Algorithm, Key Information, Implementation, Variants, In Practice, & Problems.

- Python code included in book & on companion website.

Christoph Dürr is a senior researcher at French National Center for Scientific Research (CNRS), affiliated with Sorbonne University in Paris. After a PhD in 1996 at Paris-Sud University, worked for 1 year as a postdoctoral researcher at International CS Institute in Berkeley & 1 year in School of CS & Engineering in Hebrew University of Jerusalem in Israel. Has worked in fields of quantum computation, discrete tomography (chụp cắt lớp rời rạc), & algorithmic game theory, & his current research activity focuses on algorithms & optimization. From 2007–2014: taught a preparation course for programming contests at engineering school École Polytechnique, & acts regularly as a problem setter, trainer, or competitor for various coding competitions. Love carrot cake.

Jill-Jênn Vie is a research scientist at Inria in ML. An alumnus from École normale supérieure Paris-Saclay, where founded algorithmic club of Paris-Saclay (CAPS) & coached several teams for International Collegiate Programming Contest (ICPC). Published a book in theoretical CS to help students prepare for prestigious French competitive exams e.g. *Grandes Écoles* pr *agrégation*, & directed a television show "Blame the Algorithm" about algorithms that govern our lives. He is part of advisory board of French CS Society (SIF), itself a member of International Federation for Information Processing (IFIP).

- Preface. Algorithms play an important role in our society, solving numerous mathematical problems which appear in a broad spectrum of situations. To give a few examples, think of planning taxi routes given a set of reservations (Sect. 9.1.2); assigning students to schools in a large urban school district, e.g. New York (Sect. 9.4); or identifying a bottleneck in a transportation network (Sect. 9.8). This is why job interviews in IT industry test candidates for their problem-solving skills. Many programming contests are organized by companies e.g. Google, Facebook, & Microsoft to spot gifted candidates & then send them job offers. This book will help students to develop a culture of algorithms & data structures, so that they know how to apply them properly when faced with new mathematical problems.

  Designing right algorithm to solve a given problem is only half of work; to complete job, algorithm needs to be implemented efficiently. This is why this book also emphasizes implementation issues, & provides full source code for most of algorithms presented. Have chosen Python for these implementations. What makes this language so enticing: Python allows a particularly clear & refined expression, illustrating essential steps of algorithm, without obscuring things behind burdensome notations describing data structures. Surprisingly, Python is actually possible to re-read code written several months ago & even understand it!

  Have collected 128 algorithmic problems, indexed by theme rather than by technique. Many are classic, whereas certain are atypical (khác biệt). This work should prove itself useful when preparing to solve wide variety of problems posed in programming contests e.g. ICPC, Google Code Jam, Facebook Hacker Cup, Prologin, France-ioi, etc. Hope: could serve as a basis for an advanced course in programming & algorithms, where even certain candidates for 'agrégation de mathématiques option informatique' (French competitive exam for highest teacher's certification) will find a ew original developments. Website https://tryalgo.org/, maintained by authors, contains links to code of this book + selected problems at various online contests. This allows readers to verity their freshly acquired skills.

  Hope: reader will pass many long hours tackling algorithmic problems that at 1st glance appear insurmountable (không thể vượt qua), & in end feel profound joy when a solution, especially an elegant solution, suddenly becomes apparent. Attention, it's all systems go!

- 1. Introduction.

  > You, my young friend, are going to learn to program algorithms of this book, & then go on to win programming contests, sparkle during job interviews, & finally roll up your sleeves, get to work, & greatly improve gross national product!

  Mistakenly, computer scientists are still often considered magicians of modern times. Computers have slowly crept into our businesses, our homes, & our machines, & have become important enablers in functioning of our world. However, there are many that use these devices without really mastering them, & hence, they do not fully enjoy their benefits. Knowing how to program provides ability to fully exploit their potential to solve problems in an efficient manner. Algorithms & programming techniques have become a necessary background for many professions. Their mastery allows development of creative & efficient computer-based solutions to problems encountered every day.

  This text presents a variety of algorithmic techniques to solve a number of classic problems. It describes practical situations where these problems arise, & presents simple implementations written in programming language Python. Correctly implementing an algorithm is not always easy: there are numerous traps to avoid & techniques to apply to guarantee announced running times. Examples in text are embellished with explanations of important implementation details which must be respected.

  For last several decades, programming competitions have sprung up at every level all over world, in order to promote a broad culture of algorithms. Problems proposed in these contests are often variants of classic algorithmic problems, presented as frustrating enigmas that will never let you give up until you solve them!

  - 1.1. Programming Competitions. In a programming competition, candidates must solve several problems in a fixed time. Problems are often variants of classic problems, e.g. those addressed in this book, dressed up with a short story. Inputs to problems are called *instances*. An instance can be, e.g., adjacency matrix of a graph for a shortest path problem. In general, a small example of an instance is provided, along with its solution. Source code of a solution can be uploaded to a server via

a web interface, where it is compiled & tested against instances hidden from public. For some problems code is called for each instance, whereas for others input begins with an integer indicating number of instances occurring in input. In latter case, program must then loop over each instance, solve it, & display results. A submission is accepted if it gives correct results in a limited time, on order of a few secs. Fi.1.1: Logo of ICPC nicely shows steps in resolution of a problem. A helium balloon is presented to team for each problem solved. To give a list of all programming competitions & training sites is quite impossible, & such a list would quickly become obsolete. Nevertheless, review some of most important ones.

* **ICPC.** Oldest of these competitions was founded by *Association for Computing Machinery* i 1977 & supported by them up until 2017. This contest, known as ICPC, for *International Collegiate Programming Contest*, is organized in form of a tournament. Starting point is 1 of regional competitions, e.g. *South-West European Regional Contest* (SWERC), where 2 best teams qualify for worldwide final. Particularity of this contest: each 3-person team has only a single computer at their disposal (xử lý). Have only 5 hours to solve a maximum number of problems among 10 proposed. 1st ranking criterion: number of submitted solutions accepted (i.e. tested successfully against a set of unknown instances). Next criterion: sum over submitted problems of time between start of contest & moment of accepted submission. For each erroneous submission, a penalty of 20 minutes is added.
  There are several competing theories on what ideal composition of a team is. In general, a good programmer & someone familiar with algorithms is required, along with a specialist in different domains e.g. graph theory, dynamic programming, etc. &, of course, team members need to get along together, even in stressful situations!
  For contest, each team can bring 25 pages of reference code printed in an 8-point font. They can also access online documentation of Java API & C++ standard library.

- 2. Character Strings.
- 3. Sequences.
- 4. Arrays.
- 5. Intervals.
- 6. Graphs.
- 7. Cycles in Graphs.
- 8. Shortest Paths.
- 9. Matchings & Flows.
- 10. Trees.
- 11. Sets.
- 12. Points & Polygons.
- 13. Rectangles.
- 14. Numbers & Matrices. Present a few efficient implementations of classic manipulations of numbers: arithmetical operations, evaluation of expressions, resolution of linear systems, & sequences of multiplication of matrices.

  - 14.1. GCD. Given $a, b \in \mathbb{Z}$, seek largest integer p s.t. a, b can be expressed as integer multiples of p; this is their *greatest common divisor* (GCD).

    Calculation of GCD can be implemented recursively to be very efficient. A mnemonic trick (mẹo ghi nhớ): from 2nd iteration on, arrange to always have 2nd argument smaller than 1st: $a \mod b < b$.

    ```
    def pgcd(a, b):
        return a if b == 0 else pgcd(b, a % b)
    ```

    Complexity of this division version of Euclid's algorithm is $O(\log a + \log b)$. Indeed, 1st parameter diminishes by at least a factor of 2 every 2nd iteration.

  - 14.2. Bézout Coefficients. Def: For $a, b \in \mathbb{Z}$, would like to find $u, v \in \mathbb{Z}$ s.t. $au + bv = d$ where $d = \gcd(a, b)$.

    This calculation is based on an observation similar to above. If $a = bq + r$, then $au + bv = d$ corresponds to $(bq + r)u + bv = d$, or $bu' + rv' = d$ for $u' = qu + v, v' = u \Leftrightarrow u = v', v = u' - qv'$. This calculation also terminates in $O(\log a + \log b)$ steps.

    Variant. Certain problems involve calculation of extremely large numbers, & consequently require a response modulo a large prime number p in order to test if solution is correct. Since p is prime, can easily divide by an integer a non-multiple of p: $(a, p) = 1$, hence their Bézout coefficients satisfy $au + pv = 1$, hence $au = 1 \pmod p$, & u is inverse of a. Hence, to divide by a, can instead multiply by $u \pmod p$.

  - 14.3. Binomial Coefficients. In calculation of $\binom{n}{k}$, equal to $\frac{n!}{k!(n-k)!}$, risky to calculate $n(n-1)\cdots(n-k+1)$ & $k!$ separately, given possibility of an overflow. Can exploit fact: product of i consecutive integers always contains a term divisible by i.

```
def binom(n, k):
    prod = 1
    for i in range(k):
        prod = (prod * (n - i)) // (i + 1)
    return prod
```

For most of these problems, calculation of binomial coefficients needs to be taken modulo a prime number p. Code then becomes as follows, with complexity $O(k(\log k + \log p))$, relying on calculation of Bézout coefficients.

```
def binom_modulo(n, k, p):
    prod = 1
    for i in range(k):
        prod = (prod * (n - i) * inv(i + 1, p)) % p
    return prod
```

An alternative: calculate Pascal's triangle by dynamic programming, which could be interesting if $\binom{n}{k}$ needs to be calculated for numerous pairs n, k.

◦ 14.4. Fast Exponentiation. Def: Given a, b wish to calculate a^b. Again, as result risks being enormous, often asked to perform calculation module a given integer p, but this does not change nature of problem.

Algorithm in $O(\log b)$. Naive approach performs $b-1$ multiplications by a. However, can rapidly calculate powers of a of form $a^1, a^2, a^4, a^8, \ldots$ using relation $a^{2^k} \cdot a^{2^k} = a^{2^{k+1}}$. A trick consists of combining these values according to binary decomposition of b. E.g.: $a^{13} = a^{8+4+1} = a^8 \cdot a^4 \cdot a^1$. For this calculation, suffice to generate $O(\log_2 b)$ powers of a. Implementation below is reproduced with gracious permission of Louis Abraham. At ith iteration, variable `a` contains $a_0^{2^i} \mod p$ & variable `b` contains $\lfloor \frac{b_0}{2^i} \rfloor$, where a_0, b_0: inputs to function. Thus, it suffices to test parity bit of `b` in body of loop to determine binary decomposition of b. Operation `b >>= 1` $\Leftrightarrow$ performing an integer division of b by 2.

```
def fast_exponentiation(a, b, q):
    assert a >= 0 \& b >= 0 \& q >= 1
    result = 1
    while b:
        if b % 2 == 1:
            result = (result * a) % q
        a = (a * a) % q
        b >>= 1
    return result
```

Variant. This technique can also be applied to matrix multiplications. Let A: a matrix, $b \in \mathbb{N}^\star$. Rapid exponentiation allows calculation of A^b with only $O(\log b)$ matrix multiplications.

Problem 1 (Last digit spoj:LASTDIG)**.**

Problem 2 (Tiling a grid with dominoes spoj:GNY07H])**.**

◦ 14.5. Prime numbers. *Sieve of Eratosthenes.* Given n, seek all prime numbers $< n$. Sieve of Eratosthenes is a simple method to execute this task. Begin with a list of all integers $< n$, with initially 0 & 1 marked. Then, for each $p = 2, 3, 4, \ldots, n-1$, if p is not marked, then p is prime, & in this case mark all multiples of p. Complexity of this procedure is delicate to analyze: $O(n \log \log n)$.

Implementation Details. Proposed implementation skips marking 0, 1 as well as multiples of 2. This is why iteration over p is done with a step of 2, in order to test only odd numbers.

```
def eratosthene(n):
    P = [True] * n
    answ = [2]
    for i in range(3, n, 2):
        if P[i]:
            answ.append(i)
            for j in range(2 * i, n, i):
                P[j] = False
    return answ
```

Gries–Misra Sieve. A deficiency of preceding algorithm: it marks same composite number several times. An improvement was proposed by David Gries & Jayadev Misra in 1978 [Ref] that gets around this deficiency & has a theoretical complexity of $O(n)$. Thus gain a factor of $\log \log n$. Their algorithm is not much longer but multiplicative constant in complexity is a bit larger. Our experiments have shown an improvement in execution time with, in general, a factor $\frac{1}{2}$ for interpretor `pypy`, but a deterioration by a factor 3 for interpretor `python3`.

Dexter Kozen showed: algorithm could at same time produce an array `factor` associating with every integer $2 \leq x < n$ smallest nontrivial integer factor of x. This array is very useful in generation of decomposition in prime factors of a given integer.

Algorithm is based on unique prime decomposition of a composite integer (fundamental theorem of arithmetic). Indeed, an integer y can be written in form $y = \texttt{factor}[y] \cdot x$ with $\texttt{factor}[y] \leq \texttt{factor}[x]$.

Algorithm enumerates all of composite integers by looping 1st on $x = 2, \ldots, n-1$, & then over all prime numbers p, with $p \leq \texttt{factor}[x]$, where p plays role of $\texttt{factor}[y]$ in expression $y = p \cdot x$. Algorithm is correct, since at time of processing x, all prime numbers $\leq x$ have been found, as well as smallest factor of x. As every number y between 2 & $n-1$ is examined exactly once, complexity of algorithm is $O(n)$.

```
def gries_misra(n):
    primes = []
    factor = [0] * n
    for x in range(2, n):
        if not factor[x]: # no factor found
            factor[x] = x # meaning x is prime
            primes.append(x)
        for p in primes:  # loop over primes found so far
            if p > factor[x] or p * x >= n:
                break
            factor[p * x] = p # p is smallest factor of p * x
    return primes, factor
```

Problem 3 (Spiral of primes [icpcarchive:2120]).

Problem 4 (Prime or not [spoj:PON]).

Problem 5 (Bazinga! [spoj:DCEPC505]).

- 14.6. Evaluate an Arithmetic Expression. Def: Given an expression obeying a certain fixed syntax, construct syntax (parse) tree or evaluate value of expression, see Fig. 14.1: Syntax tree associated with expression $2 + (3 \times 4)/2$.

  Approaches. p. 218
- 14.7. System of Linear Equations.
- 14.8: Multiplication of a Matrix Sequence.

- 15. Exhaustive Search.
- 16. Conclusion.
  - 16.1. Combine Algorithms to Solve a Problem. At times, problem may require to find an optimal value. Can this value be found by dichotomy (sự phân đôi)? If it lies between 0 & N & if condition verifies a certain property, this adds only a factor $\log N$ to complexity. This is why it is always profitable to know how to efficiently code a binary search. Try your skills on this problem, 1 of most difficult: it can be arranged [kattis:itcanbearranged].

    Do you have to count number of subsequences, or some other combinatorial structure, module some large prime number? Then, a recurrence relation by dynamic programming can be combined with replacing every division by a multiplication with inverse modulo p (Sect. 14).

    Is a greedy algorithm sufficient, or are we faced with a problem of dynamic programming? What is recurrence? Is recursive structure of problem at hand a special case of some known problem? It is only by solving lots & lots of problems that you can build & hone these reflexes.

    Finally, size of instances can sometimes provide a clue as to expected complexity, see table in Sect. 1.4.
  - 16.2. For Further Reading. A selection of works completing subjects treated in this book.
    - An essential ref for fundamental algorithms: substantial *Introduction to Algorithms* by T.H. Cormen, C. E. Leiserson, R. L. Rivest, & C. Stein, MIT Press.
    - For a variety of more specialized algorithms, see *Encyclopedia of Algorithms*, a collective work edited by Ming-Yang Kao, Springer Verlag, 2e, 2016.
    - Flow algorithms are studied from top to bottom in *Network Flows: Theory, Algorithms, & Applications* by R.K. Ahuja, T. L. Magnanti, & J. B. Orlin, Pearson, 2013.
    - A good introduction to algorithms for problems in geometry is *Computational Geometry: Algorithms & Applications* by M. de Berg, O. Cheung, M. van Kreveld, M. Overmars, Springer Verlag 2008.
    - If would like to learn more about tricky manipulations with binary representation of integers, a delightful ref: *Hacker's Delight* by Henry S. Warren, Jr, Addison-Wesley, 2013.
    - A very good introduction to programming in Python: *Think Python: How to Think Like a Computer Scientist* by Allen Downey, O'Reilly Media, 2015.

* Other highly appreciated books include *Python Essential Reference* by David M. Beazley, Pearson Education, 2009 & *Python Cookbook* by Brian K. Jones, O'Reilly Media, 2013.
* 2 texts more adapted to preparation for programming contests: *Competitive Programming* by Steven & Felix Halim, Lulu, 2013 & *The Algorithm Design Manual* by Steven S. Skiena, Springer Verlag, 2009.

- 16.3. Rendez-vous on tryalgo.org. This book is accompanied by a website https://tryalgo.org/, where code of all Python programs described here can be found + test data, training exercises & solutions that, of course, must not be read before having attempted problems. Package `tryalgo` is available on GitHub & PyPI under MIT license, & can be installed under Python 2 or 3 via `pip install tryalgo`.

```
>>> import tryalgo
>>> help(tryalgo)   # for the list of modules
>>> help(tryalgo.arithm) # for a particular module
```

3.2 [Laa20; Laa24]. Antti Laaksonen. Guide to Competitive Programming: Learning & Improving Algorithms Through Contests

- Amazon review. This textbook features new material on advanced topics, e.g. calculating Fourier transforms, finding minimum cost flows in graphs, & using automata in string problems. Critically, text accessibly describes & shows how CP is a proven method of implementing & testing algorithms, as well as developing computational thinking & improving both programming & debugging skills.

  Topics & features.

  1. Introduces dynamic programming & other fundamental algorithm design techniques, & investigates a wide selection of graph algorithms
  2. Compatible with IOI Syllabus, yet also covering more advanced topics, e.g. maximum flows, Nim theory, & suffix structures
  3. Provides advice for students aiming for IOI contest
  4. Surveys specialized algorithms for trees, & discusses mathematical topics that are relevant in CP
  5. Examines use of Python language in CP
  6. Discusses sorting algorithms & binary search, & examines a selection of data structures of C++ standard library
  7. Explores how GenAI will impact on future of field
  8. Covers such advanced algorithm design topics as bit-parallelism & amortized analysis, & presents a focus on efficiently processing array range queries
  9. Describes a selection of more advanced topics, including square-root algorithms & DP optimization

  Fully updated, expanded & easy to follow, this core textbook/guide is an ideal reference for all students needing to learn algorithms & to practice for programming contests. Knowledge of programming basics is assumed, but prev background in algorithm design or programming contests is not necessary. With its breadth of topics, examples, & references, book is eminently suitable for both beginners & more experienced readers alike.

- 1. Introduction. This chap shows what CP is about, outlines contents of book, & discusses additional learning resources.
  - Sect. 1.1 goes through elements of CP, introduces a selection of popular programming contests, & gives advice on how to practice CP.
  - Sect. 1.2 discusses goals & topics of this book, & briefly describes contents of each chap.
  - Sect. 1.3 presents CSES Problem set, which contains a collection of practice problems. Solving problems while reading book is a good way to learn CP.
  - Sect. 1.4 discusses other books related to CP & design of algorithms.

  - 1.1. What is CP? CP combines 2 topics: design of algorithms & implementation of algorithms.

    Design of Algorithms. Core of CP is about inventing efficient algorithms that solve well-defined computational problems. Design of algorithms requires problem solving & mathematical skills. Often a solution to a problem is a combination of well-known methods & new insights.

    Mathematics play an important role in CP. Actually, there are no clear boundaries between algorithm design & mathematics. This book has been written so that not much background in mathematics is needed. Appendix of book reviews some mathematical concepts that are used throughout book, e.g. sets, logic & functions, & appendix can be used as a reference when reading book.

    Implementation of Algorithms. In CP, solutions to problems are evaluated by testing an implemented algorithm using a set of test cases. Thus, after coming up with an algorithm that solves problem, next step: correctly implement it, which requires good programming skills. CP greatly differs from traditional software engineering: programs are short (usually at most some hundreds of lines), they should be written quickly, & not need to maintain them after contest.

At tie of writing this book, C++ is clearly most popular language in CP. Java & Python also have some popularity, & other languages are little used. E.g., in 2023, C++ was used in 91% of CSES Problem Set submissions, Java in 3% & Python in 5%.

Many people regard C++ as best choice for a competitive programmer. In particular, almost all top competitive programmers use C++. Benefits of using C++ are that it is a very efficient language & its standard library contains a large collection of data structures & algorithms.

All examples programs in this book are written in C++, except for Chap. 16 that discusses Python. Data structures & algorithms available in C++ standard library are often used in book. Programs follow C++11 standard, which can be used in most contests nowadays. If cannot program in C++ yet, now is a good time to start learning.

* 1.1.1. Programming Contests.
* 1.1.2. Tips for Practicing.

- 1.2. About This Book.
- 1.3. CSES Problem Set.
- 1.4. Other Resources.

- 2. Programming Techniques. This chap presents some of features of C++ programming language that are useful in CP, & gives examples of how to use recursion & bit operations in programming.
  - Sect. 2.1 discusses a selection of topics related to C++, including input & output methods, working with numbers, & how to shorten code.
  - Sect. 2.2 focuses on recursive algorithms. 1st learn an elegant way to generate all subsets & permutations of a set using recursion. After this, use backtracking to count number of ways to place n non-attacking queens on $n \times n$ chessboard.
  - Sect. 2.3 discusses basics of bit operations, & shows how to use t hem to represent subsets of sets.
  - 2.1. Language Features. A typical C++ code template for CP looks like this:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    // solution comes here
}
```

`#include` line at beginning of code is a feature of `g++` compiler that allows us to include entire standard library. Thus, not needed separately include libraries e.g. *iostream, vector, algorithm*, but rather they are available automatically.

`using` line declares: classes & functions of standard library can be used directly in code. Without `using` line would have to write, e.g., `std::cout`, but now it suffices to write `cout`.

Code can be compiled using command:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

This command produces a binary file `test` from source code `test.cpp`. Compiler follows C++11 standard `-std=c++11`, optimizes code `-O2` & shows warnings about possible errors `-Wall`.

* 2.1.1. Input & Output. In most contests, standard streams are used for reading input & writing output. In C++, standard streams are `cin` for input & `cout` for output. Also C functions, e.g. `scanf, printf` can be used.
  Input for program usually consists of numbers & strings separated with spaces & newlines. they can be read from `cin` stream as follows:

```
int a, b;
string x;
cin >> a >> b >> x;
```

This kind of code always works, assuming: there is at least 1 space or newline between each element in input. E.g., above code can read both following inputs:

```
123 456 monkey
123    456
monkey
```

`cout` stream is used for output as follows:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Input & output is sometimes a bottleneck in program. Following lines at beginning of code make input & output more efficient:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Note: newline `"\n"` works faster than `endl`, because `endl` always causes a flush operation, see, e.g., Geeks4Geeks/What does buffer flush means in C++?
C functions `scanf, printf` are an alternative to C++ standard streams. They are usually slightly faster, but also more difficult to use. Code reads 2 integers from input:

```
int a, b;
scanf("%d %d", &a, &b);
```

Code prints 2 integers:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Sometimes program should read a whole input line, possibly containing spaces. This can be accomplished by using `getline` function:

```
string s;
getline(cin, s);
```

If amount of data is unknown, following loop is useful:

```
while (cin >> x) {
    // code
}
```

This loop reads elements from input one after another, until there is no more data available in input.
In some contest systems, files are used for input & output. An easy solution for this: write code as usual using standard streams, but add following lines to beginning of code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, program reads input from file `input.txt` & writes output to file `output.txt`.

* 2.1.2. Working with Numbers.
* 2.1.3. Shortening Code.

- 3. Efficiency.
- 4. Sorting & Searching.
- 5. Data Structures.
- 6. Dynamic Programming.
- 7. Graph Algorithms.
- 8. Algorithm Design Topics.
- 9. Range Queries.
- 10. Tree Algorithms.
- 11. Mathematics.
- 12. Advanced Graph Algorithms.
- 13. Geometry.
- 14. String Algorithms.
- 15. Additional Topics.
- 16. Python in Competitive Programming. While C++ is predominant language in CP, Python has also gained some popularity in recent years. Even if one usually uses C++, Python has some useful features that can be worth learning. In this chap, take a look at Python language from a CP perspective.

- Sect. 16.1 shows how to solve an example CP problem using Python. After that, go through some Python features: handling input & output, working with numbers & generating combinations of objects.
- Sect. 16.2 deals with Python data structures that are useful in CP: list structures, hash structures & priority queues. Also discuss differences between Python & C++ when using data structures.
- Sect. 16.3 presents 2 scenarios where a natural C++ solution: use binary search tree data structures. Python standard library does not have such data structures, & have to find alternative ways to solve problems.
- Sect. 16.4 discusses use of recursive functions in Python. See how to increase default recursion depth limit & how to implement dynamic programming solutions in Python.
- Sect. 16.5 studies efficiency of Python through experiments. Both compare efficiency of 2 Python implementations (CPython & PyPy) & efficiency between Python & C++.
- Sect. 16.6 shows how Python can be used as a tool for generating tests, stress testing solutions & implementing an algorithm for finding polynomials.

- 16.1. Introduction. At beginning of this book, created a C++ solution to *Weird Algorithm* problem from CSES Problem Set. Now solve problem in Python.

  When solved problem in C++, had to carefully select type of variable n so that it is large enough to hold all intermediate values during calculation. In Python, do not have to think about this because built-in Python integers can contain arbitrary large values. Other than that, there are no big differences when implementing solution in C++ & Python.

  * 16.1.1. Input & Output. Python function `input` reads a line from standard input & returns line as a string. E.g., following code reads 2 lines from standard input. 1st line is converted to an integer, & 2nd line is stored as a string.

    ```
    n = int(input())
    s = input()
    ```

    If a line consists of several values separated by spaces, can use `split` function to convert them to a list:

    ```
    t = input().split()
    ```

    Using above code, each value in list is a string. To convert each value to an integer, can use following code which uses list comprehension syntax to construct a list of integers:

    ```
    t = [int(x) for x in input().split()]
    ```

    Another approach: use `map` function for integer conversion & `list` function to create a list from a map object.

    ```
    t = list(map(int, input().split()))
    ```

    `print` function writes a line to standard output. E.g., code writes 3 lines:

    ```
    print(a)
    print(b)
    print(c)
    ```

    Code writes each value on same line, separated by spaces:

    ```
    print(a, b, c)
    ```

    `end, sep` parameters can be used to change newline & separator strings when `print` function. By default, `end` is `"\n"` & `sep` is `" "`. Code writes values on same line, separated by spaces:

    ```
    print(a, end=" ")
    print(b, end=" ")
    print(c)
    ```

    Code writes each value on a separate line:

    ```
    print(a, b, c, sep="\n")
    ```

  * 16.1.2. Working with Numbers. In Python, built-in integers can contain arbitrarily large values. E.g., following code prints integer 1337^{13}:

    ```
    print(1337**13)
    ```

    Output of code:

    ```
    436222733061138473758786649126561772142 97
    ```

Thus, convenient to use Python when necessary to handle integers that do not fit into 64 bit or 128 bit integer types in C++. [However, in some recent Python versions, conversions between large integers & their string representations are restricted. E.g., not possible by default to print an integer that has > 4300 digits. Purpose of this change: prevent denial of service attacks in Python applications, because such conversions are slow. `sys.set_int_max_str_digits` function can be used to increase limit.]
Operator `/` always produces a floating point number, even if both numbers are integers. Operator `//` can be used for integer division.

```
print(3 / 2) # 1.5
print(3 // 2) # 1
```

`pow` function can be used to efficiently calculate value of an expression $a^b \mod c$. E.g., code prints value of $999^{10^{>}6} \mod 123$.

```
print(pow(999, 10**6, 123)) # 42
```

Module `math` contains some useful functions for integer calculations. [Some of these functions are not available in older Python versions.] Functions `gcd, lcm` calculate greatest common divisor & lowest common multiple for a list of numbers. Function `factorial` returns factorial of a number, & function `comb` can be used to calculate binomial coefficients.

```
import math
print(math.gcd(8, 12)) # 4
print(math.gcd(8, 12, 6)) # 2
print(math.lcm(8, 12)) # 24
print(math.factorial(5)) # 120
print(math.comb(5, 3)) # 10
```

Module `fractions` provides a way to perform exact calculations with fractions. E.g., code creates fractions $\frac{1}{2}$ & $\frac{5}{7}$:

```
from fractions import Fraction
a = Fraction(1, 2)
b = Fraction(5, 7)
print(a) # 1/2
print(b) # 5/7
print(float(a)) # 0.5
print(float(b)) # 0.7142857142857143
```

Fractions are automatically shown in reduced form:

```
print(Fraction(1, 2)) # 1/2
print(Fraction(2, 4)) # 1/2
print(Fraction(3, 6)) # 1/2
```

Mathematical operators can be used for calculating with fractions:

```
a = Fraction(1, 2)
b = Fraction(5, 7)
print(a + b) # 17/14
print(a * b) # 5/14
print(a < b) # True
```

∗ 16.1.3. Generating Combinations. Module `itertools` can be used to generate combinations of objects. Module includes following functions:

- Function `permutations` generate all permutations of input sequence. E.g., permutations of $[1, 2, 3]$ are $(1, 2, 3), (1, 3, 2), (2, 1, 3), ($
- Function `combinations` generates all subsequences of input sequence that have k elements. E.g., when input sequence is $[1, 2, 3]$ & $k = 2$, combinations are $(1, 2), (1, 3), (2, 3)$. Each subsequence corresponds to a subset of size k.
- Function `product` generates all sequences of length k where each element comes from input sequence. E.g., when input sequence if $[1, 2, 3]$ & $k = 2$, sequences are $(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)$.
- Function `combinations_with_replacements` generates all sequences of length k where each element comes from input sequence & order of elements is same as input sequence. E.g., when input sequence is $[1, 2, 3]$ & $k = 2$, combinations are $(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)$.

Code demonstrates how to use functions:

```
import itertools
s = [1, 2, 3]
k = 2
print(list(itertools.permutations(s)))
print(list(itertools.combinations(s, k)))
```

```
print(list(itertools.product(s, repeat=k)))
print(list(itertools.combinations_with_replacement(s, k)))
```

Output of code:

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2),
(3, 2, 1)]
[(1, 2), (1, 3), (2, 3)]
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1),
(3, 2), (3, 3)]
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

◦ 16.2. Data Structures. Python standard library has several data structures that are useful in CP. In this sect, discuss some of those data structures & also highlight some differences between Python & C++ data structures.
1 difference between Python & C++: in Python, syntax `a = b` only copies reference to data structure & does not copy contents of data structure like in C++. Code demonstrates this:

```
a = [1, 2, 3]
b = a
a.append(4)
print(a) # [1, 2, 3, 4]
print(b) # [1, 2, 3, 4]
```

Here variables `a, b` point to same list & when an element is added through `a`, it will also be visible through `b`.
Same applies to function calls: if modify a list given to a function as a parameter, changes will be visible outside function. In following code, function adds a new element to a list that is given as a parameter.

```
def test(x):
    x.append(4)
a = [1, 2, 3]
test(a)
print(a) # [1, 2, 3, 4]
```

Another difference: Python has 2 types of data structures: mutable & immutable data structures. Mutable data structures can be modified using methods & operators but immutable data structures cannot be modified.
E.g., a list is a mutable data structure because can use `[]` syntax to modify list:

```
x = [1, 2, 3]
x[1] = 5
print(x) # [1, 5, 3]
```

However, strings & tuples are immutable & there are no methods or operators that could be used to modify them. E.g., following code does not work because it is not allowed to modify a string:

```
x = "abc"
x[1] = "e" # TypeError
print(x)
```

∗ 16.2.1. List Structures. In Python, a list is a dynamic array where elements can be efficiently added & removed at end of list. Python list structure corresponds to C++ vector structure.
Lists can be created using `[]` syntax. Following code creates a list with 3 elements:

```
t = [1, 2, 3]
```

Another way to create a list: use method `append` that adds a new element at end of list:

```
t = []
t.append(1)
t.append(2)
t.append(3)
print(t) # [1, 2, 3]
```

Method `pop` removes & returns last element of list:

```
t = [1, 2, 3]
print(t.pop()) # 3
print(t) # [1, 2]
```

Elements of list can be accessed using `[]` syntax:

```
t = [1, 2, 3]
print(t[1]) # 2
t[1] = 5
print(t[1]) # 5
```

Sorting lists. There are 2 ways to sort a list. 1st way: use `sort` method:

```
t = [3, 2, 1]
t.sort()
print(t) # [1, 2, 3]
```

Another way: use `sorted` function which creates a new sorted list without modifying original list:

```
t = [3, 2, 1]
print(sorted(t)) # [1, 2, 3]
print(t) # [3, 2, 1]
```

Deques. Python also has a deque data structure that allows efficient insertion & deletion of elements both at beginning at end of list. It has 2 special methods `appendleft, popleft` that modify beginning of list.
Code shows how a deque can be used:

```
from collections import deque
d = deque()
d.append(1)
d.append(2)
d.appendleft(3)
print(d) # [3, 1, 2]
d.pop()
print(d) # [3, 1]
d.popleft()
print(d) # [1]
```

In Python, deques are implemented as linked lists & not possible to efficiently access their elements using `[]` syntax. This differs from C++ where deques are implemented as dynamic arrays.

∗ 16.2.2. Hash Structures. Python has 2 useful data structures that are based on hash tables: a set & a dictionary. They correspond to C++ `unordered_set, unordered_map` data structures.
Sets. A set maintains a collection of elements. It provides efficient methods `add, remove` for element insertion & removal. In addition, operator `in` can be used to efficiently check if a set contains an element.
Code shows how to use set data structure:

```
s = set()
s.add(1)
s.add(2)
s.add(3)
print(s) # {1, 2, 3}
print(2 in s) # True
s.remove(2)
print(2 in s) # False
```

Each element can appear at most once in a set:

```
s = set()
s.add(5)
s.add(5)
s.add(5)
print(s) # {5}
```

Dictionaries. A dictionary consists of key-value pairs which can be efficiently accessed through keys. Dictionaries can be created using `{}` syntax, & values can be accessed using `[]` syntax.
Code shows how to use dictionary data structure:

```
d = {}
d["monkey"] = 1
d["banana"] = 2
d["harpsichord"] = 3
print("banana" in d) # True
print(d["banana"]) # 2
```

Unlike in C++, there are no default values. E.g., following code does not work because dictionary does not have key "monkey".

```
d = {}
print(d["monkey"]) # KeyError: 'monkey'
```

However, there is another data structure `defaultdict` that has default values for missing keys. E.g., can define a dictionary with type `int` that works as follows:

```
from collections import defaultdict
d = defaultdict(int)
print(d["monkey"]) # 0
d["monkey"] += 1
print(d["monkey"]) # 1
```

Only immutable values, e.g. members, strings & tuples, can be used as keys in Python sets & dictionaries. E.g., code does not work because a list is not immutable:

```
s = set()
s.add([1, 2, 3]) # TypeError
```

∗ 16.2.3. Priority Queues. Module `heapq` has functions that perform binary heap operations on a list. 1st element of list is minimum element. Function `heappush` adds an element to heap, & function `heappop` removes & returns minimum element. Using those functions, can use lists as priority queues.
Code shows how functions can be used:

```
from heapq import heappush, heappop
q = []
heappush(q, 2)
heappush(q, 1)
heappush(q, 4)
heappush(q, 3)
print(q[0]) # 1
heappop(q)
print(q[0]) # 2
```

Both functions `heappush, heappop` work in $O(\log n)$ time. In addition, function `heapify` can be used to convert a list to a heap in $O(n)$ time. Function can be used as follows:

```
from heapq import heapify
q = [2, 1, 4, 3]
heapify(q)
print(q) # [1, 2, 4, 3]
```

◦ 16.3. Coping Without Binary Search Trees. Python standard library does not have binary search tree data structures. Thus, there are no equivalents for C++ data structures `set, map` in Python.

Using binary search tree data structures, can maintain a set where possible to efficiently find minimum & maximum elements & also process queries like "what is smallest element larger than x" or "what is largest element smaller than x". In Python, such features are not available in standard library.

Fortunately, even if it would seem at 1st glance: a problem requires binary search trees, can often solve problem using an alternative way that only requires tools included in Python standard library, e.g. sorting, hash structures & priority queues.

∗ 16.3.1. Minimum Queries. 1st consider situation where would like to use a data structure that has following operations:

· add an element to set

· remove an element from set

· find minimum element in set

Turn out: can create such a data structure using 2 Python data structures: a set & a priority queue. When an element is added, add it to both data structures. When an element is removed, remove it only from set because not possible to remove arbitrary elements from a priority queue.
Most interesting operation in data structure is finding minimum element. Priority queue can be used to find minimum element, but it can contain elements that have already been removed. For this reason, as long as minimum element in priority queue is not in set anymore, remove it also from priority queue. Then, when find an element that is also in set, return it.
Can implement operations as follows:

```
def add(x):
```

```
        s.add(x)
        heappush(q, x)
    def remove(x):
        s.remove(x)
    def find_min():
        while q[0] not in s:
            heappop(q)
        return q[0]
```

Using this implementation, `find_min` function may need to remove a large number of elements from priority queue before returning actual minimum element. However, every element that is added to set is removed at most once. For this reason, each function works efficiently on average.

Note: previously used as similar trick for implementing Dijkstra's algorithm using a C++ priority queue (Sect. 7.3.2).

∗ 16.3.2. Example Problem. While can support minimum & maximum queries using a combination of a set & a priority queue, there is no easy way to process queries like "what is smallest element $> x$". However, if would need such queries in a problem, there may still be a way to avoid them using an alternative approach.

Consider problem from CSES Problem Set:

Problem 6. *There are n concert tickets available, each with a certain price. Then, m customers arrive, one after another. Each customer announces the maximum price they are willing to pay for a ticket, & after this, they will get a ticket with nearest possible price such that it does not exceed the maximum price.*

Input. *1st input line contains $n, m \in \mathbb{N}$: number of tickets & number of customers. The next line contains n integers $h_1, h_2, \ldots, h_n$: the price of each ticket. The last line contains m integers $t_1, t_2, \ldots, t_m$: the maximum price for each customer in the order they arrive.*

Output. *Print, for each customer, the price that they will pay for their ticket. After this, ticket cannot be purchased again. If a customer cannot get any ticket, print -1.*

Constraints. $1 \leq m, n \leq 2 \cdot 10^5$, $1 \leq h_i, t_i \leq 10^9$.

In C++, could use a `multiset` for solving problem. 1st, could add all ticket prices to set & then efficiently find ticket price for each customer. To find ticket price, could use `upper_bound` function to find smallest ticket price that is too large & then choose prev element in set which is desired ticket price.

To solve problem in Python, have to design another solution that does not require maintaining a set of ticket prices with efficient queries for finding ticket price for a customer. Turn out: this problem is easier to solve if change order customers are processed & create an offline solution.

Idea: create a list of events. There are 2 types of events: (1) a customer wants to buy a ticket whose price is $\leq x$ & (2) there is a new ticket available with price x. Sort list in descending order by ticket price, & then process events 1 by 1. In addition, maintain a priority queue that contains customers who have already requested a ticket. Customers are sorted by arrival time in priority queue.

When process an event of type 1, add customer to priority queue. When process an event of type 2, remove customer with minimum arrival time from priority queue & give ticket to that customer. If there are no customers in priority queue, nobody will get ticket. After have processed all events, can report ticket price for each customer.

There is a linear number of events & can efficiently process each event using priority queue, which results in an efficient algorithm.

◦ 16.4. Recursive Functions. Consider recursive Python function that computes factorial of n:

```
def factorial(n):
    if n == 0:
        return 1
    return factorial(n - 1) * n
```

For small numbers, function works fine:

```
print(factorial(0)) # 1
print(factorial(2)) # 2
print(factorial(5)) # 120
print(factorial(9)) # 362880
```

However, cannot compute factorial of 1000:

```
print(factorial(1000)) # RecursionError
```

Reason for this: maximum recursion depth in Python is quite small by default. Can increase limit by using `sys.setrecursionlimit` functions as follows:

```
import sys
sys.setrecursionlimit(5000)
print(factorial(1000)) # 402387260077093773543702433923...
```

∗ 16.4.1. Dynamic Programming. Can often use dynamic programming to make a recursive function efficient. A recursive function that calculates Catalan numbers (Sect. 11.2.2):

```
def catalan(n):
    if n == 0:
        return 1
    s = 0
    for i in range(n):
        s += catalan(i) * catalan(n - i - 1)
    return s
```

For small numbers, function works fine:

```
print(catalan(2)) # 2
print(catalan(3)) # 5
print(catalan(5)) # 42
```

However, for larger numbers function is slow because number of recursive calls is too large.

```
print(catalan(100)) # too slow
```

A way to make function efficient using DP:

```
def catalan(n, d={}):
    if n == 0:
        return 1
    if n not in d:
        s = 0
        for i in range(n):
            s += catalan(i) * catalan(n - i - 1)
        d[n] = s
    return d[n]
```

Function uses a dictionary to store its return values & it makes recursive calls only once for each parameter. After this modification, function is efficient:

```
print(catalan(100)) # 896519947090131496687170070074...
```

Note how each function call uses same dictionary to store results, which is given as a parameter. In Python, if a function parameter has a default value which creates a data structure, each function call that uses default value uses same data structure.
In above situation this feature is useful but it can sometimes cause confusion. E.g., consider following code:

```
def test(t=[]):
    t.append(1)
    print(t)
test() # [1]
test() # [1, 1]
test() # [1, 1, 1]
```

One could think: each function call adds an element to an empty list. However, in reality each function call adds an element to same list.

∗ 16.4.2. Cache Decorator. In Python there is also a built-in way to store function results & create efficient dynamic programming solutions. Module `functools` has a decorator `cache` that can be used as follows [`cache` decorator was recently added to Python. Older Python versions have a similar decorator `lru_cache(maxsize=None)`.]

```
import functools
@functools.cache
def catalan(n):
    if n == 0:
        return 1
    s = 0
    for i in range(n):
        s += catalan(i) * catalan(n - i - 1)
```

```
    return s
```

Using decorator, function return values for different parameters are stored automatically & previously calculated value is returned if function is called again with same parameter.

- 16.5. Efficiency. In this sect, compare efficiency of 2 Python implementations (CPython & PyPy) & C++. CPython is standard Python implementation which is most common way to execute Python code. PyPy is an alternative Python implementation which includes a just-in-time compiler & is often faster.
  - 16.5.1. Finding Primes. In 1st experiment, count number of primes between 2 & n using sieve of Eratosthenes. Python code used in experiment:

```
sieve = [0]*(n+1)
count = 0
for i in range(2, n+1):
    if sieve[i]:
        continue
    count += 1
    for j in range(2*i, n+1, i):
        sieve[j] = 1
print(count)
```

Corresponding code in C++:

```
vector<int> sieve(n+1);
int count = 0;
for (int i = 2; i <= n; i++) {
    if (sieve[i]) continue;
    count++;
    for (int j = 2*i; j <= n; j += i) {
        sieve[j] = 1;
    }
}
cout << count << "\n";
```

Table 16.1: Results of prime counting experiment shows results of experiment. It can be seen: PyPy & C++ are much faster than CPython in large tests. However, difference between PyPy & C++ is quite small.

  - 16.5.2. Counting Permutations.

- 17. Preparing for IOI.
- 18. Future of Competitive Programming.
- Appendix: Mathematical Background.

3.3 [WW16]. Yonghui Wu, Jiande Wang. Data Structure Practice for Collegiate Programming Contests & Education

- Amazon review. Combining knowledge with strategies, *Data Structure Practice for Collegiate Programming Contests & Education* presents 1st comprehensive book on data structure in programming contests. This book is designed for training collegiate programming contest teams in nuances of data structure & for helping college students in computer-related majors to gain deeper understanding of data structure.

  Based on successful experiences in many world-level contests, book includes 204 typical problems & detailed analyzes selected from ACM International Collegiate Programming Contest & other major programming contests since 1990. It is divided into 4 sects that focus on:

  - Fundamental programming skills
  - Experiments for linear lists
  - Experiments for trees
  - Experiments for graphs

  Each chap contains a set of problems & includes hints. Book also provides test data for most problems as well as sources & IDs for online judgments that help with improving programming skills.

  Introducing a multi-options model & considerations of context, *Data Structure Practice for Collegiate Programming Contests & Education* encourages students to think creatively in solving programming problems. By taking readers through practical

contest problems from analysis to implementation, it provides a complete source for enhancing understanding & polishing skills in programming.

Author Author. YONGHUI WU was coach of Fudan University programming contest teams 2001–2011. Under his guidance, Fudan University qualified for Association for Computing Machinery International Collegiate Programming Contest (ACM-ICPC) World Finals every year, winning 3 medals during that span: bronze medal in 2002, silver medal in 2005, & bronze medal in 2010. Since 2012, he has published a series of books for programming contests & education. He is now chair of ICPC Asia Programming Contest 1st Training Committee.

JIAN-DE WANG is a famous coach for Olympiad in Informatics in China. Under his guidance, his students have won 7 gold medals, 3 silver medals, & 2 bronze medals for China in International Olympiad in Informatics. He has published 24 books for programming contests.

- Preface. Since 1990s, ACM International Collegiate Programming Contest (ACM-ICPC) has become a worldwide programming contest. Every year, $>$ 10000 students & $>$ 1000 universities participate in local contests, preliminary contests, & regional contests all over world. In meantime, programming contests' problems from all over world can be gotten, analyzed, & solved by us. These contest problems can be used not only for programming contest training, but also for education.

  In our opinion, not only a programming contestant's ability, but also a computer student's ability is based on his or her programming knowledge system & programming strategies for solving problems. Programming knowledge system can be summarized as a famous formula: algorithms + data structures = programs. It is also foundation for knowledge system of CS & engineering. Strategies solving problems are strategies for data modeling & algorithm design. When data models & algorithms for problems are not standard, what strategies should take to solve these problems?

  Based on ACM-ICPC, published a series of books, not only for systematic programming contest training, but also for better polishing computer students' programming skill, using programming contests' problems: "Data Structure Experiment: For Collegiate Programming Contest & Education", "Algorithm Design Experiment: For Collegiate Programming Contest & Education", & "Programming Strategies Solving Problems" in Mainland China. & traditional Chinese versions for "Data Structure Experiment: For Collegiate Programming Contest & Education" & "Programming Strategies Solving Problems" were also published in Taiwan.

  "Data Structure Practice: For Collegiate Programming Contests & Education" is English version for "Data Structure Experiment: For Collegiate Programming Contest & Education". There are 4 sects, 14 chaps, & 200 programming contest problems in this book. Sect. I, "Fundamental Programming Skills", focused on experiments & practices for simple computing, simple simulation, & simple recursion, for students just learning programming languages. Sect. II, "Experiments for Linear Lists", Sect. III, "Experiments for Trees", & Sect. IV "Experiments for Graphs", focus on experiments & practices for data structure.

  Characteristics of book:

  1. Book's outlines are based on outlines of data structures. Programming contest problems & their analyzes & solutions are used as experiments. For each chap, there is a "Problems" sect to let students solve programming contests' problems, & hints for these problems are also shown.
  2. Problems in book are all selected from ACM-ICPC regional & world finals programming contests, universities' local contests, & online contests, & from 1990–now.
  3. Not only analyzes & solutions or hints to problems are shown, but also test data for most of problems are provided. Sources & IDs for online judges for these problems are also given. They can help readers better & more easily polish their programming skills.

  Therefore, book can be used not only as an experiment book, but also for systematic programming contests' training.

  Online judge systems for problems in this book:

  - Peking University Online Judge System [POJ] http://poj.org/
  - Zhejiang University Online Judge System [ZOJ] http://acm.zju.edu.cn/onlinejudge/
  - UVA Online Judge System [UVA] http://uva.onlinejudge.org/
  - http://livearchive.onlinejudge.org/
  - Ural Online Judge System [Ural] http://acm.timus.ru/
  - SGU Online Judge System [SGU] http://acm.sgu.ru/

- SECT. I: FUNDAMENTAL PROGRAMMING SKILLS. Programming language is an introductory course of data structures & algorithms. This course enables students to program by programming languages. Programming languages, data structures, & algorithm designs are skills that computer students must polish. Therefore, polishing fundamental programming skills is 1st sect for this book. There are 3 chaps in Sect. I covering: 1. Computing. 2. Simulation. 3. Recursion. These 3 chaps are not only a review of programming languages, but also an introductory course on data structure.

  - 1. Practice for Simple Computing. Pattern of a programming contest problem is input–process–output. A problem for simple computing is a problem whose process is simple. For such a problem, should only consider optimizing process & dealing with input & output correctly. Goals of Chap. 1:

1. Students master C/C++ or Java programming language.
2. Students become familiar with online judge systems & programming environments.
3. Students begin to learn how to transfer a practical problem into a computing process, implement computing process by a program, & debug program to pass all test cases.

"God is in details." In Chap. 1, problems are relatively simple. Should notice formats of input & output, precision, & time complexity. Therefore, following topics are discussed in this chap:

1. Programming style
2. Multiple test cases
3. Precision of real numbers
4. Improving time complexity by dichotomy (sự phân đôi, not đĩ chó Tố My)

Normally, a complex problem consists of several subproblems for simple computing. "Even longest journey begins with a single step." Polishing programming skills should begin with solving simple computing problems.

∗ 1.1. Improving Programming Style. A program's writing style is not only for its visual sense, but also for examining program & debugging its errors. A program's style also shows whether its programming idea is clear. Hard to say which kind of programming style is good, but there are some rules for programming style. They are discussed in following experiments.

Problem 7 (Financial Management)**.** LARRY *graduated this year & finally has a job. He's making a lot of money, but somehow never seems to have enough.* LARRY *has decided: need to get a hold of his financial portfolio & solve his financial problems. 1st step: figure out what's been going on with his money.* LARRY *has his bank account statements & wants to see how much money he has. Help* LARRY *by writing a program to take his closing balance from each of past 12 months & calculate his average account balance.*

- **Input.** *Input will be 12 lines. Each line will contain closing balance of his bank account for a particular month. Each number will be positive & displayed to penny. No dollar sign will be included.*
- **Output.** *Output will be a single number, average (mean) of closing balances for 12 months. It will be rounded to nearest penny, preceded immediately by a dollar sign, & followed by end of line. There will be no other spaces or characters in output.*
- *Source: ACM Mid-Atlantic United States 2001.*
- *IDs for online judges: POJ 1004, ZOJ 1048, UVA 2362.*

Analysis. Problem's pattern, input–process–output, is very simple: 1st, income of 12 months `a[0..11]` is input by a for statement `for(i=0; i < 12; i++)`, & total income $sum = \sum_{i=0}^{11} a[i]$ is calculated. Then average monthly income $avg = \frac{sum}{12}$ is calculated. Finally, avg is output in accordance with problem's output format.

From above program, can get: 1st, input & output of program must meet formats for input & output. In this problem, each input number will be positive & displayed to penny, & output will be rounded to nearest penny, preceded immediately by a dollar sign & followed by an end of line. If program doesn't meet formats for input & output, it will be judged as wrong answer.

2nd, a program should be readable. Style of a program should be serration based on a logical level.

Finally, program annotations should be given

∗ 1.2. Multiple Test Cases. Financial management problem (Sect. 1.1.1) has only 1 test case. In order to guarantee correctness of a program, for most problems there are multiple test cases. In some circumstances, number of test cases is given; in other circumstances, number of test cases isn't given, but mark of input end is given.

Problem 8 (Doubles)**.** *As part of an arithmetic competency program, your students will be given randomly generated lists of 2–15 unique positive integers & asked to determine how many items in each list are twice some other item in same list. Will need a program to help you with grading. This program should be able to scan lists & output correct answer for each one. E.g., given list* `1 4 3 2 9 7 18 22`*, your program should answer 3, as 2 is twice 1, 4 is twice 2, & 18 is twice 9.*

Input. *Input file will consist of 1 or more lists of numbers. There will be 1 list of numbers per line. Each list will contain from 2–15 unique positive integers. No integer will be* > 99*. Each line will be terminated with integer 0, which is not considered part of list. A line with single number* -1 *will mark end of file. Example input below shows 3 separate lists. Some lists may not contain any doubles.*

Output. *Output will consist of 1 line per input list, containing a count of items that are double some other item.*

Source: ACM Mid-Central United States 2003.

IDs for online judges: POJ 1552, ZOJ 1760, UVA 2787.

Analysis. There are multiple test cases for problem. Therefore, a loop statement is used to deal with multiple test cases. Loop enumerates every test case. -1 marks end of input. Therefore, -1 is end condition of loop. In loop statement, there are 2 steps:

1. A loop inputs a test case into array `a` & accumulates number of elements n in test case. 0 marks end of test case.
2. A double loop enumerates all pairs of $a[i]$ & $a[j]$ $(0 \le i < n-1, i+1 \le j < n)$ in test case & determines whether `(a[i]*2 == a[j] || a[j]*2 == a[i])` holds.

In this problem, number of test cases & size of a test case are unknown. Normally, a double-loop statement is used for program structure: outer loop is used to enumerate every test case, & inner loop is used to deal with a test case.

In some problems, if size of test data is larger, all test cases are dealt with by same method, & result area is known, its time complexity can be improved by an offline method. 1st, all solutions within specified range are calculated & stored in a constant array. Then program deals with constant array directly for each test case. It can avoid duplication of computing.

Problem 9 (Sum of consecutive prime numbers)**.** *Some positive integers can be represented by a sum of 1 or more consecutive prime numbers. How many such representations does a given positive integer have? E.g., integer* 53 *has 2 representations* $5 + 7 + 11 + 13 + 17$ *&* 53*. Integer* 41 *has 3 representations:* $2 + 3 + 5 + 7 + 11 + 13$*,* $11 + 13 + 17$*, &* 41*. Integer* 3 *has only 1 representation, which is* 3*. Integer* 20 *has no such representations. Note: summands must be consecutive prime numbers, so neither* $7 + 13$ *nor* $3 + 5 + 5 + 7$ *is a valid representation for integer* 20*. Your mission: write a program that reports number of representations for given positive integer.*

Input. *Input is a sequence of positive integers, each in a separate line. Integers are between* 2 *&* 10000*, inclusive. End of input is indicated by a* 0*.*

Output. *Output should be composed of lines each corresponding to an input line, except last* 0*. An output line includes number of representations for input integer as sum of 1 or more consecutive prime numbers. No other character should be inserted in output.*

Source: ACM Japan 2005.
IDs for online judges: POJ 2739, UVA 3399.

Analysis. Because program needs to deal with consecutive prime numbers for each test case, & upper limit of prime numbers is 10000, offline method can be used to solve problem. 1st, all prime numbers < 10001 are obtained & stored in array `prime[1 .. total]` in ascending order. Then deal with test cases 1 by 1: Suppose input number is n; sum of consecutive prime numbers is `cnt`; number of representation for `cnt == n` is `ans`. A double loop is used to get number of representations for n:

- Outer loop i: `for(int i = 0; n >= prime[i]; i++)` enumerates all possible minimum `prime[i]`.
- Inner loop j: `for(int j = i; j < total && cnt < n; j++), cnt += prime[j]`, is to calculate sum of consecutive prime numbers. If `cnt` $\geq n$, then loop ends, & if `cnt == n`, then number of representations is `ans++`.

When outer loop ends, `ans` is solution to test case.

- 2. Simple Simulation. In real world, there are many problems that we can solve by simulating their processes. Such problems are called simulation problems. For these problems, solution procedures & rules are showed in problem descriptions. Programs must simulate procedures or implement rules based on descriptions.

  Normally there are 2 kinds of simulations: stochastic simulation & process simulation.

  Problems for stochastic simulation show or imply probabilities. Programmers make use of random functions & round functions to set random value for a range, making random value meet probability as a parameter. Then programmers design algorithm by simulating mathematical model. Because of uncertainty, there are fewer problems for stochastic simulation in programming contests.

  Problems for process simulation require programmers to design parameters for mathematical models, & to observe changes of states caused by parameters. Programmers design algorithms based on process simulation. Programs depend entirely on authenticity & correctness of process simulation without any uncertainties.

  This chap focuses on process simulation. There are 3 kinds of process simulation:

  1. Simulation of direct statement
  2. Simulation by sieve method
  3. Simulation by construction

  - 2.1. Simulation of Direct Statement. For problems for simulation of direct statement, programmers are required to solve them by strictly following rules showed in problem's descriptions. Programmers must read such problems carefully & simulate processes based on descriptions. A problem for simulation of direct statement gets harder as number of rules increases. It causes amount of code to grow & become more illegible.

- 3. Simple Recursion.
- Summary of Sect. I.

- SECT. II: EXPERIMENTS FOR LINEAR LISTS.
  - 4. Linear Lists Accessed Directly.
  - 5. Applications of Linear Lists for Sequential Access.
  - 6. Generalized List Using Indexes.
  - 7. Sort of Linear Lists.
  - Summary of Sect. II.
- SECT. III: EXPERIMENTS FOR TREES.
  - 8. Programming by Tree Structure.
  - 9. Applications of Binary Trees.

- 10. Applications of Classical Trees.

- SECT. IV: EXPERIMENTS FOR GRAPHS.
  - 11. Applications of Graph Traversal.
  - 12. Algorithms of Minimum Spanning Trees.
  - 13. Algorithms of Best Paths.
  - 14. Algorithms of Bipartite Graphs & Flow Networks.
  - Summary of Sect. IV.

3.4 [WW18]. Yonghui Wu, Jiande Wang. Algorithm Design Practice for Collegiate Programming Contests & Education

- Amazon review. This book can be used as an experiment & reference book for algorithm design courses, as well as a training manual for programming contests. It contains 247 problems selected from ACM-ICPC programming contests & other programming contests. There's detailed analysis for each problem. All problems, & test datum for most of problems will be provided online. Content will follow usual algorithms syllabus, & problem-solving strategies will be introduced in analyses & solutions to problem cases. For students in computer-related majors, contestants & programmers, this book can polish their programming & problem-solving skills with familarity of algorithms & mathematics.

  Author Author. Dr. Yonghui Wu serves as Associate Professor at the school of CS in Fudan University, China. He acted coach of Fudan University Programming Contest teams from 2001 to 2011. Under his guidance Fudan University was qualified for ACM ICPC World Finals every year & won 3 medals (bronze medal in 2002, silver medal in 2005, & bronze medal in 2010) in ACM ICPC World Finals. Since 2012, he has published a series of books for programming contest & education in China covering datastructures, algorithms & strategies, which are warmly welcomed not only in Mainland China, but also in Taiwan, Hongkong, Singapore, & other Chinese speaking areas. He just published his 1st book's English version: *Data Structure Practice: for Collegiate Programming Contests & Education* with CRC Press. Since 2013, he has been giving lectures in Oman, Taiwan & the United States for programming contest training. He is currently chair of ICPC Asia Programming Contest 1st Training Committee.

- Preface. Programming contests are contests solving problems by programming. Starting in 1990s, ACM International Collegiate Programming Contest (ACM-ICPC) has become a worldwide programming contest. Every year, 6 continents, over 110 countries, 50000 students, 5000 coaches, & 3000 universities participate in ACM-ICPC local contests, preliminary contests, & regional contests all over world. Alongside, some international programming contests, e.g. Google Code Jam, TopCoder Open Algorithm, Facebook Hacker Cup, Internet Problem Solving Contest (IPSC), & so on, are held every year. Programmers from all over world, in addition to students, can participate in these contests through Internet.

  Based on these programming contests, programming contests' problems from all over world can be obtained, analyzed, & solved by students. These contest problems can be used not only for programming contest training, but also for education.

  In our opinion, not only programming contestants' ability to solve problems, but also computer students' programming skills are based on their programming knowledge system & programming strategies for solving problems. Programming knowledge system can be summarized as "Algorithms + Data Structures = Programs." Also foundation for knowledge system of CS & engineering. Strategies for solving problems are strategies for data modeling & algorithm design. When data models & algorithms for problems are not standard, need to take some strategies to solve these problems.

  Based on these facts, published a series of books, not only for systematic programming contest training, but also for polishing computer students' programming skill better, using programming contests problems: *Data Structure Experiment for Collegiate Programming Contest & Education, Algorithm Design Experiment for Collegiate Programming Contest & Education, & Programming Strategies Solving Problems* in Mainland China. & traditional Chinese version for *Data Structure Experiment for Collegiate Programming Contest & Education, & Programming Strategies Solving Problems* were also published in Taiwan. In 2016, 1st book's English version *Data Structure Practice: for Collegiate Programming Contest & Education* was published by CRC Press.

  Algorithm Design Practice for Collegiate Programming Contest & Education is English version for *Algorithm Design Experiment for Collegiate Programming Contest & Education*. There are 9 chaps & 247 programming contest problems in this book.

  - Chap. 1: "Practice for Ad Hoc Problems", focuses on solving problems that there are no classical algorithms to solve. There are 2 methods to solve such problems: mechanism analysis method & statistical analysis method.
  - Chap. 2: "Practice for Simulation Problems", experiments & practices for simulation problems are shown. In problem descriptions, solution procedures or rules are shown. Simulation problems are solved by implementing rules or simulating solution procedures.
  - Chap. 3: "Practice for Number Theory", Chap. 4: "Practice for Combinatorics", & Chap. 8: "Practice for Computational Geometry", introduce mathematical background for number theory, combinatorics, & computational geometry, resp., & then show problems solved by mathematical methods. Greedy algorithms & DP are used to solve optimization problems.
  - Chap. 5: "Practice for Greedy Algorithms", & Chap. 6, "Practice for DP", introduce greedy algorithms & DP resp., & show problems solved by greedy algorithms & DP.

- Chap. 7: "Practice for Advanced Data Structures", describes using suffix arrays, segment trees, & some graph algorithms to solve problems. Search technologies are fundamental to CS & technology.
- Chap. 9: "Practice for State Space Search", describes implementation of state space search through solving contest problems.

Feature of book:

1. Book's outlines are based on outlines of algorithms. Programming contest problems & their analyses & solutions are used as experiments. For each chap, there is a "Problems" sect to let students solve programming contests' problems, & hints for these problems are also included.
2. Problems in book are all selected from ACM-ICPC regional & world finals programming contests, universities' local contests, & online contests, from 1990 to now.
3. Not only analyses & solutions, or hints to problems are shown, but also test data for most of problems are provided. Sources & IDs for Online Judge for these problems are also provided. This can help readers polish their programming skills better & more easily.

Book can be used not only as an experiment book, but also for training for systematic programming contests.

- 1. Practice for Ad Hoc Problems. Ad hoc means "for special purpose or end presently under consideration." (cho mục đích đặc biệt hoặc mục đích hiện đang được xem xét) there are no classical algorithms that can solve these ad hoc problems. Programmers need to design specific algorithms to solve ad hoc problems. There are 2 strategies to design algorithms for solving ad hoc problems: mechanism analysis & statistical analysis.

  To solve an ad hoc problem, need to see past its appearance & understand its essence. In this chap, 2 kinds of analyses solving ad hoc problems are shown: Mechanism Analysis & Statistical Analysis.

  - 1.1. Solving Problems by Mechanism Analysis. Mechanism analysis examines characteristics & internal mechanisms of an object to find a mathematical representation of problem. Therefore, key to mechanism analysis is mathematical modeling. Solving problems by mechanism analysis is a top-down method.
  - 1.2. Solving Problems by Statistical Analysis. Unlike mechanism analysis, statistical analysis begins with a partial solution to problem, & overall global solution is found based on analyzing partial solution. Solving problems by statistical analysis is a bottom-up method.
- 2. Practice for Simulation Problems.
- 3. Practice for Number Theory.
- 4. Practice for Combinatorics.
- 5. Practice for Greedy Algorithms.
- 6. Practice for Dynamic Programming.
- 7. Practice for Advanced Data Structures.
- 8. Practice for Computational Geometry.
- 9. Practice for State Space Search.
- Bibliography.

4 Conda

4.1 Should I use Anaconda Distribution or Miniconda?

https://docs.anaconda.com/distro-or-miniconda/. Both Anaconda Distribution & Miniconda installers include conda package & environment manager, but how you plan to use software will determine which installer you want to choose. Table of Cfs.
What's your experience level?

- *I'm just starting out & don't know what packages I should use.* Install Anaconda Distribution! It includes over 300 standard DS & ML packages, which will give you a kickstart in your development journey.
- *I don't have much experience with the command line.* Install Anaconda Distribution! The install includes Anaconda Navigator, a desktop application that is built on top of conda. You can use Navigator's graphical user interface (GUI) to create environments, install packages, & launch development applications like Jupyter Notebooks & Spyder. For more information on Navigator, see Getting started with Navigator.
- *I know exactly what packages I want to use & I don't want a large download.* Install Miniconda! Miniconda is a minimal installer that only includes a very small installation of Anaconda's distribution–conda, Python, & a few other packages. For more information, see Miniconda documentation.
- *I only use the command line.* Install Miniconda or Anaconda Distribution! Both installations include conda, the command line package & environment manager. For more information on conda, see conda Getting Started page or download conda cheatsheet.

4.2 Anaconda Distribution

The Most Trusted Distribution for Data Science. Anaconda Distribution is a Python/R data science distribution that contains:

- conda - a package & environment manager for your command line interface
- Anaconda Navigator - a desktop application built on conda, with options to launch other development applications from your managed environments.
- > 300 automatically-installed packages that work well together out of box
- Access to Anaconda Public Repository, with 8000 open-source DS & ML packages

4.2.1 System requirements

- License: Free for individuals & small organizations (< 200 employees). A paid license is required for larger organizations & anyone embedding or mirroring Anaconda. See TOS for details.
- OS: Windows 10 or later, 64-bit macOS 10.15+ (for Intel) or 64-bit macOS 11.1+ (for Apple Silicon), or Linux, including Ubuntu, RedHat, CentOS 7+, & others.
- If OS is older than what is currently supported, can find older versions of Anaconda installers in archive that might work for you. See Using Anaconda on older OSs for version recommendations.
- System architecture: Windows - 64-bit x86; MacOS - 64-bit x86 or Apple Silicon (ARM64); Linux - 64-bit x86, 64-bit aarch64 (AWS Graviton2), or s390x (Linux on IBM Z & LinuxONE).
- `linux-aarch64` Miniconda installer requires `glibc >=2.26` & thus will not work with CentOS 7, Ubuntu 16.04, or Debian 9 ("stretch").
- `linux-aarch64` package builds might not be compatible with certain Raspberry Pi setups, as Anaconda uses compiler options that target server-class Neoverse N1/N2 microarchitecture.
- Minimum 5 GB disk space to download & install.

Remark 1. *Best practice to install Anaconda Distribution for local user, which does not require administrator permissions & is most robust type of installation. However, if have administrator permissions, can install Anaconda Distribution system-wide.*

4.2.2 Installing Anaconda Distribution

Provides instructions for installing Anaconda Distribution on Windows, macOS, & Linux.

Remark 2. *If prefer an installation without extensive collection of packages included in Anaconda Distribution, install Miniconda instead.*

Miniconda is a free, miniature installation of Anaconda Distribution that includes only conda, Python, packages they both depend on, & a small number of other useful packages.

- Basic install instructions.

  Remark 3. *If you've installed multiple versions of Anaconda Distribution, system defaults to using most current version, as long as you haven't altered default install path.*

  Linux installer. Anaconder Navigator is included with Anaconda Distribution by default. However, following extended dependencies might be required for certain versions of Anaconda Distribution in order to use Anaconda Navigator (& other GUI packages) with Linux:

  - **Debian.**

```
nqbh@nqbh-dell:~$ sudo apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2 libxrandr2 libxss1 libxcur
sudo: unable to resolve host nqbh-dell: Name or service not known
[sudo] password for nqbh:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Package libasound2 is a virtual package provided by:
liboss4-salsa-asound2 4.2-build2020-1ubuntu3
libasound2t64 1.2.11-1build2 (= 1.2.11-1build2)
You should explicitly select one to install.

Package libgl1-mesa-glx is not available, but is referred to by another package.
```

```
This may mean that the package is missing, has been obsoleted, or
is only available from another source

E: Package 'libgl1-mesa-glx' has no installation candidate
E: Unable to locate package libegl1-mesa
E: Package 'libasound2' has no installation candidate

nqbh@nqbh-dell:~$ curl -O https://repo.anaconda.com/archive/Anaconda3-2024.10-1-Linux-x86_64.sh
```

Ensure that you are downloading an installer that is compatible with your operating system!

(Recommended) Verify the integrity of your installer to ensure that it was not corrupted or tampered with during download. To ensure that your downloaded installer has not been tampered with or corrupted, generate its SHA-256 hash value & compare it to the official hash provided in archive. E.g.,

```
nqbh@nqbh-dell:~$ shasum -a 256 ~/Anaconda3-2024.10-1-Linux-x86_64.sh
3ba0a298155c32fbfd80cbc238298560bf69a2df511783054adfc151b76d80d8  /home/nqbh/Anaconda3-2024.10-1-Linux-x8
```

Note: generated SHA-256 hash value from the output. Visit repo.anaconda.com/archive to find the official SHA-256 hash for your installer. Compare the hash values. If they match, the installer is safe to use.

Install Anaconda Distribution by running 1 of following commands (depending on your Linux architecture):

```
bash ~/Anaconda3-2024.10-1-Linux-x86_64.sh
```

Press return to review license agreement & hold return to scroll. Enter yes to agree to the license agreement.

```
Downloading \& Extracting Packages:

Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish to update your shell profile to automatically initialize conda?
This will activate conda on startup \& change the command prompt when activated.
If you'd prefer that conda's base environment not be activated on startup,
run the following command when conda is activated:

conda config --set auto_activate_base false

You can undo this by running `conda init --reverse $SHELL`? [yes|no]
[no] >>>

You have chosen to not have conda modify your shell scripts at all.
To activate conda's base environment in your current shell session:

eval "$(/home/nqbh/anaconda3/bin/conda shell.YOUR_SHELL_NAME hook)"

To install conda's shell functions for easier access, first activate, then:

conda init

Thank you for installing Anaconda3!
```

Press return to accept default install location (`PREFIX=/Users/<USER>/anaconda3`), or enter another file path to specify an alternate installation directory. Installation might take a few minutes to complete. Choose an initialization options:

* Yes - conda modifies your shell configuration to initialize conda whenever you open a new shell & to recognize conda commands automatically.
* No - conda will not modify your shell scripts. After installation, if you want to initialize, you must do so manually. For more information, see Manual Shell Initialization.

Installer finishes & displays, "Thank you for installing Anaconda3!" Close & re-open your terminal window for the installation to fully take effect, or use the following command to refresh the terminal, depending on your shell: `source ~/.bashrc`.

Manual Shell Initialization. Once installation has successfully completed, initialize your shell by running following command:

```
# Replace <PATH_TO_CONDA> with the path to your conda install
source <PATH_TO_CONDA>/bin/activate
conda init

qbh@nqbh-dell:~$ source ~/anaconda3/bin/activate
(base) nqbh@nqbh-dell:~$ conda init
no change     /home/nqbh/anaconda3/condabin/conda
no change     /home/nqbh/anaconda3/bin/conda
no change     /home/nqbh/anaconda3/bin/conda-env
no change     /home/nqbh/anaconda3/bin/activate
no change     /home/nqbh/anaconda3/bin/deactivate
no change     /home/nqbh/anaconda3/etc/profile.d/conda.sh
no change     /home/nqbh/anaconda3/etc/fish/conf.d/conda.fish
no change     /home/nqbh/anaconda3/shell/condabin/Conda.psm1
no change     /home/nqbh/anaconda3/shell/condabin/conda-hook.ps1
no change     /home/nqbh/anaconda3/lib/python3.12/site-packages/xontrib/conda.xsh
no change     /home/nqbh/anaconda3/etc/profile.d/conda.csh
modified      /home/nqbh/.bashrc

==> For changes to take effect, close \& re-open your current shell. <==

(base) nqbh@nqbh-dell:~$
```

Remark 4. *You can also control whether or not your shell has the base environment activated each time it opens. Following commands only work if conda has been initialized 1st:*

* *Activated by default:* `conda config --set auto_activate_base True`
* *Not activated by default:* `conda config --set auto_activate_base False`

Verify install. Anaconda Navigator, Graphical User Interface (GUI) for conda, should automatically open after successful installation of Anaconda Distribution. If it does not, verify your installation by opening the application manually, or by using the CLI:

* Opening Navigator manually: Open Navigator by running the following command: `anaconda-navigator`
* Using conda to verify manually: Access the CLI for your operating system. You should see (base) in the command line prompt. This tells you that you're in your base conda environment. To learn more about environments, see Environments. Run any conda command, e.g.:
  - `conda list` - Displays a list of packages installed in your active environment & their versions.
  - `anaconda-navigator` - Opens Anaconda Navigator.

- Advanced install options. For more advanced installation instructions, such as installing with silent mode, installing on older operating systems, or multi-user installs, see Advanced installation.
- Other ways to get Anaconda or Miniconda. For instructions on using Anaconda Docker images or the Anaconda Cloudera Distributed Hadoop cluster, see Applications/Integrations.

4.2.3 Advanced installation

4.2.4 Getting started

This document is here to help you get started with Anaconda Distribution, which includes conda `https://conda.io/en/latest/`, Anaconda Navigator, & over 300 scientific & machine learning packages.

- Should I use Anaconda Navigator or conda? Anaconda Navigator is a desktop application that is included with every installation of Anaconda Distribution. It is built on top of conda, open-source & environment manager, & allows you to manage your packages & environments from a graphical user interface (GUI). This is especially helpful when you're not comfortable with command line.

  A command line interface (or CLI) is a program on your computer that processes text commands to do various tasks. Conda is a CLI program, which means it can only be used via command line. On Windows computers, Anaconda recommends: use Anaconda Prompt CLI to work with conda. MacOS & Linux users can use their built-in command line applications.

  Remark 5. *If installed Miniconda instead of Anaconda Distribution, Anaconda Navigator is not included. Use command* `conda install anaconda-navigator` *to manually install Navigator onto your computer.*

- Free Anaconda Learning course - Get Started with Anaconda. Learn to use Anaconda Navigator to launch an application. Then, create & run a simple Python program with Spyder & Jupyter Notebook. Watch short training videos on Anaconda Learning to get up & running with Jupyter Notebook & JupyterLab, along with several other popular integrated development environments (IDEs): Anaconda Learning.

4.2.5 Anaconda Distribution release notes

4.2.6 Uninstalling Anaconda Distribution

4.3 Miniconda

https://docs.anaconda.com/miniconda/. Miniconda is a free, miniature installation of Anaconda Distribution that includes only conda, Python, packages they both depend on, & a small number of other useful packages.

If need more packages, use `conda install` command to install from thousands of packages available by default in Anaconda's public repo, or from other channels, like conda-forge or bioconda.

4.3.1 Latest Miniconda installer links

For latest Miniconda installers for Python 3.12, go to https://www.anaconda.com/download/. Miniconda installers are on same page as Anaconda Distribution installers, past registration.

For a list of Miniconda hashes & an archive of Miniconda versions, including installers or older versions of Python, see https://repo.anaconda.com/miniconda.

4.3.2 Quick command line install

5 LaTeX

5.1 Overleaf/glossaries

“

” – Overleaf/glossaries

6 Linux

Resources – Tài nguyên.

1. [Sho19]. William Shotts. *The Linux Command Line: A Complete Introduction.*

I used SUSE & OpenSUSE in WIAS Berlin but I do not like them & the like, so I go back to Ubuntu.

7 Programming

7.1 C/C++

Resources – Tài nguyên.

1. [Ngọ02]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C.*
2. [Ngọ03]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++.*
3. [Str13]. Bjarne Stroustrup. *The C++ Programming Language.*
4. [Str18]. Bjarne Stroustrup. *A Tour of C++.*

7.2 Pascal

Resources – Tài nguyên.

1. [Ngọ08]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal.*
2. [Ngọ09]. Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal.*
3. [DT06]. Lê Văn Doanh, Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal.*

7.3 Python

Resources – Tài nguyên.

1. [Dức22]. Nguyễn Tiến Dức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python.*
2. [Huy24]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1.*
3. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_2**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 2.*
4. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_3**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 3.*
5. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_4**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 4.*
6. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_5**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 5.*
7. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_6**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 6.*
8. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_7**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 7.*

8 Software

8.1 FeNiCS

Resources – Tài nguyên.

1. [Dok20]. Jørgen S. Dokken. *Automatic shape derivatives for transient PDEs in FEniCS & Firedrake.*
2. [LL16]. Hans Petter Langtangen, Anders Logg. *Solving PDEs in Python.*

8.2 Firedrake

8.3 Fireshape

Resources – Tài nguyên.

1. [PW20]. Alberto Paganini, Florian Wechsung. *Fireshape Documentation, Release 0.0.1.*
2. [PW21]. Alberto Paganini, Florian Wechsung. *Fireshape: a shape optimization toolbox for Firedrake.*

8.4 Git

Resources – Tài nguyên.

1. [CS14]. Scott Chacon, Ben Straub. *Pro Git.*

8.5 Gmsh

Resources – Tài nguyên.

1. [GR09]. Christophe Geuzaine, Jean-François Remacle. *Gmsh: A 3D finite element mesh generator with built-in pre- & post-processing facilities.*

8.6 OpenFOAM

Resources – Tài nguyên.

1. There are 3 variants of OpenFOAM:
   (a) OpenFOAM.com: Commercial.
   (b) OpenFOAM.org: Open-source with a large community.
   (c) Extended OpenFOAM.
2. [GW22]. Christopher Greenshields, Henry Weller. *Notes on Computational Fluid Dynamics: General Principles.*
3. [TN13]. M. Towara, U. Naumann. *A Discrete Adjoint Model for OpenFOAM.*

8.7 ParMooN

Resources – Tài nguyên.

1. [Wil+17]. Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, Volker John. *ParMooN – a modernized program package based on mapped finite elements.*

8.8 SU2

8.9 Sublime Text

Resources – Tài nguyên.

1. [Bos14]. Wes Bos. *Sublime Text Power User: A Complete Guide.*

2. [Pel13]. Dan Peleg. *Mastering Sublime Text*

9 SymPy

9.1 Matrices

A module that handles matrices. Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

- Matrices (linear algebra).
  - Creating Matrices. Linear algebra module is designed to be as simple as possible. 1st, import & declare 1st `Matrix` object:

```
>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1  0  0]
[       ]
[0  0  0]
>>> Matrix([M, (0, 0, -1)])
[1  0  0 ]
[        ]
[0  0  0 ]
[        ]
[0  0  -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
[ ]
[2]
[ ]
[3]
```

    In addition to creating a matrix from a list of appropriately-sized lists &/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values & dimension inputs:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])
[1  2  3]
[       ]
[4  5  6]
```

    More interesting (& useful), is ability to use a 2-variable function (or `lambda`) to create a matrix. Create an indicator function which is 1 on diagonal & then use it to make identity matrix:

```
>>> def f(i,j):
...     if i == j:
...         return 1
...     else:
...         return 0
>>> Matrix(4, 4, f)
[1  0  0  0]
```

```
[          ]
[0  1  0  0]
[          ]
[0  0  1  0]
[          ]
[0  0  0  1]
```

Use `lambda` to create a 1-line matrix with 1's in even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1  0  1  0]
[          ]
[0  1  0  1]
[          ]
[1  0  1  0]
```

There are also a couple of special constructors for quick matrix construction: `eye`: identity matrix, `zeros, ones` for matrices of all 0s & 1s, resp., & `diag` to put matrices or elements along diagonal:

```
>>> eye(4)
[1  0  0  0]
[          ]
[0  1  0  0]
[          ]
[0  0  1  0]
[          ]
[0  0  0  1]
>>> zeros(2)
[0  0]
[    ]
[0  0]
>>> zeros(2, 5)
[0  0  0  0  0]
[              ]
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[       ]
[1  1  1]
[       ]
[1  1  1]
>>> ones(1, 3)
[1  1  1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1  0  0]
[       ]
[0  1  2]
[       ]
[0  3  4]
```

- Basic Manipulation. While learning to work with matrices, choose one where entries are readily identifiable. 1 useful thing to know: while matrices are 2D, storage is not & so it is allowable – though one should be careful – to access entries as if they were a 1D list.

```
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

More standard entry access is a pair of indices which will always return value at corresponding row & column of matrix"

```
>>> M[1, 2]
6
>>> M[0, 0]
1
```

```
>>> M[1, 1]
5
```

Since this is Python, also able to slice submatrices; slices always give a matrix in return, even if dimension is 1×1:

```
>>> M[0:2, 0:2]
[1  2]
[    ]
[4  5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In 2nd example above notice: slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) 1st row/column is 0.

Cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1  2  3]
[       ]
[4  5  6]
```

Slicing an empty matrix works as long as use a slice for coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of 1 object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice: changing `M2` didn't change `M`. Since can slice, can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1   2   3   4 ]
[               ]
[5   6   7   8 ]
[               ]
[9   10  11  12]
[               ]
[13  14  15  16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1   2   3   0 ]
[               ]
[5   6   7   8 ]
[               ]
[9   10  0   12]
[               ]
```

```
[13  14  15  16]
```

as well as assign slices:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1   2   3  4]
[           ]
[5   6   7  8]
[           ]
[9   10  0  0]
[           ]
[13  14  0  0]
```

All standard arithmetic operations are supported:

```
>>> M = Matrix(([1,2,3],[4,5,6],[7,8,9]))
>>> M - M
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
>>> M + M
[2   4   6 ]
[          ]
[8   10  12]
[          ]
[14  16  18]
>>> M * M
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[  ]
[29]
[  ]
[47]
>>> M**2
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)
>>> M
[4  5  6]
[       ]
[7  8  9]
>>> M.col_del(1)
>>> M
[4  6]
[    ]
[7  9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
```

```
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall: `row_del(), col_del()` operations don't return a value – they simply change matrix object. Can also "glue" together matrices of appropriate size:

```
>>> M1 = eye(3)
>>> M2 = zeros(3, 4)
>>> M1.row_join(M2)
[1  0  0  0  0  0  0]
[                   ]
[0  1  0  0  0  0  0]
[                   ]
[0  0  1  0  0  0  0]
>>> M3 = zeros(4, 3)
>>> M1.col_join(M3)
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
[       ]
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
```

- Operations on entries. Not restricted to have multiplication between 2 matrices:

```
>>> M = eye(3)
>>> 2*M
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
>>> 3*M
[3  0  0]
[       ]
[0  3  0]
[       ]
[0  0  3]
```

but can also apply functions to our matrix entries using `applyfunc()`. Declare a function that double any input number. Then apply it to 3×3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
```

If want to extract a common factor from a matrix, can do so by applying `gcd` to data of matrix:

```
>>> from sympy.abc import x, y
>>> from sympy import gcd
>>> m = Matrix([[x, y], [1, x*y]]).inv('ADJ'); m
[  x*y       -y    ]
[--------  --------]
[ 2         2      ]
[x *y - y  x *y - y]
[                  ]
[  -1         x    ]
[--------  --------]
[ 2         2      ]
[x *y - y  x *y - y]
>>> gcd(tuple(_))
   1
--------
 2
x *y - y
>>> m/_
[x*y  -y]
[       ]
[-1   x ]
```

1 more useful matrix-wide entry application function: substitution function. Declare a matrix with symbolic entries then substitute a value. Remember: can substitute anything – even another symbol!

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x  0  0]
[       ]
[0  x  0]
[       ]
[0  0  x]
>>> M.subs(x, 4)
[4  0  0]
[       ]
[0  4  0]
[       ]
[0  0  4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y  0  0]
[       ]
[0  y  0]
[       ]
[0  0  y]
```

- Linear algebra. Now have basics out of way, see what can do with actual matrices. Determinant:

```
>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0
```

Another common operation is inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but can specify it be done by LU decomposition as well:

```
>>> M2.inv()
```

```
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M2.inv(method="LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M.inv(method="LU")
[-3/14  1/14  1/2 ]
[                 ]
[-1/28  5/28  -1/4]
[                 ]
[ 3/7   -1/7   0  ]
>>> M * M.inv(method="LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
```

Can perform a QR factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[  ___     ___      ___ ]
[\/ 6   -\/ 3    -\/ 2  ]
[-----  -------  -------]
[  6       3        2   ]
[                       ]
[  ___     ___      ___ ]
[\/ 6   -\/ 3     \/ 2  ]
[-----  -------   ----- ]
[  6       3        2   ]
[                       ]
[  ___     ___          ]
[\/ 6    \/ 3           ]
[-----   -----      0   ]
[  3       3            ]
>>> R
[            ___        ]
[  ___  4*\/ 6       ___]
[\/ 6   -------  2*\/ 6 ]
[          3            ]
[                       ]
[          ___          ]
[        \/ 3           ]
[  0     -----      0   ]
[          3            ]
[                       ]
[                   ___ ]
[  0       0      \/ 2  ]
>>> Q*R
[1  1  1]
[       ]
[1  1  3]
[       ]
[2  3  4]
```

In addition to solvers in `solver.py` file, can solve system $A\mathbf{x} = \mathbf{b}$ by passing vector $\mathbf{b}$ to matrix A's `LUsolve` function. Here

cheat a little choose A & $\mathbf{x}$ then multiply to get $\mathbf{b}$. Then can solve for $\mathbf{x}$ & check it's correct:

```
>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]
```

STOP HERE

- Matrix Kind.
- Dense Matrices.
- Sparse Matrices.
- Sparse Tools.
- Immutable Matrices.
- Matrix Expressions.
- Matrix Normal Forms.

10 Miscellaneous

11 Wikipedia

11.1 Wikipedia/abstraction (computer science)

"In software engineering & computer science, *abstraction* is the process of generalizing concrete details, e.g. attributes, away from the study of objects & systems to focus attention on details of greater importance. Abstraction is a fundamental concept in computer science & software engineering, especially within the object-oriented programming paradigm. E.g.:

- the usage of abstract data types to separate usage from working representations of data within programs.
- the concept of functions or subroutines which represent a specific way of implementing control flow;
- the process of reorganizing common behavior from groups of non-abstract classes into abstract classes using inheritance & subclasses, as seen in object-oriented programming languages.

11.1.1 Rationale

> "The essence of abstraction is preserving information that is relevant in a given context, & forgetting information that is irrelevant in that context." – John V. Guttag

Computing mostly operates independently of the concrete world. The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentrating on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's 10th rule is an aphorism on how such an architecture is both inevitable & complex.

Language abstraction is a central form of abstraction in computing: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer language* from the machine language to the assembly language & the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues e.g. in scritping languages & domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include subroutines, modules, polymorphism, & software components. Some other abstractions such as software design pattens & architectural styles remain invisible to a translator & operate only in the design of a system.

Some abstractions try to limit the range of concepts a programmer needs to be aware of, by completely hiding the abstractions they are built on. The software engineer & writer Joel Spolsky has criticized these efforts by claiming that all abstractions are *leaky* – that they can never completely hide the details below; however, this does not negate the usefulness of abstraction.

Some abstractions are designed to inter-operate with other abstractions – e.g., a programming language may contain a foreign function interface for making calls to the lower-level language.

11.1.2 Abstraction features

Programming languages. Main article: Wikipedia/programming language. Different programming languages provide different types of abstraction, depending on the intended applications for the language. E.g.:

- In OOP languages e.g. C++, Object Pascal, or Java, the concept of *abstraction* has itself become a declarative statement – using the syntax `function(parameters) = 0;` (in C++) or the keywords `abstract` & `interface` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.
- Functional programming languages commonly exhibit abstractions related to functions, e.g. lambda abstractions (making a term into a function of some variable) & higher-order functions (parameters are functions).
- Modern members of the Lisp programming language family e.g. Clojure, Scheme, & Common Lisp support macro systems to allow syntactic abstraction. Other programming languages such as Scala also have macros, or very similar metaprogramming features (e.g., Haskell has Template Haskell, & OCaml has MetaOCaml). These can allow a programmer to eliminate boilerplate code, abstract away tedious function call sequences, implement new control flow structures, & implement Domain Specific Languages (DSLs), which allow domain-specific concepts to be expressed in concise & elegant ways. All of these, when used correctly, improve both the programmer's efficiency & the clarity of the code by making the intended purpose more explicit. A consequence of syntactic abstraction is also that any Lisp dialect & in fact almost any programming language can, in principle, be implemented in any modern Lisp with significantly reduced (but still nontrivial in most cases) effort when compared to "more traditional" programming languages such as Python, C, or Java.

Specification methods. Main article: Wikipedia/formal specification. Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE, CASL);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages Main article: Wikipedia/specification language. Specification languages generally rely on abstractions of 1 kind or another, since specifications are typically defined earlier in a project, (& at a more abstract level) than an eventual implementation. The UML specification language, e.g., allows the definition of *abstract* classes, which in a waterfall project, remain abstract during the architecture & specification phrase of the project.

11.1.3 Control abstraction

Main article: Wikipedia/control flow. Programming languages offer control abstraction as 1 of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from 1 location of the memory to another location & producing the sum of 2 sequences of bits. Programming languages allow this to be done in the higher level. E.g., consider this statement written in a Pascal-like fashion: `a := (1 + 2) * 5`. To a human, this seems a fairly simple & obvious calculation. However, the lower-level steps necessary to carry out this evaluation, & return the value 15, & then assign that value to the variable `a`, are actually quite subtle & complex. The values need to be converted to binary representation (often a much more complicated task than one would think) & the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of 1 register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of 15 to the variable labeled `a`, so that `a` can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label & the resultant location in physical or virtual memory, storing the binary representation of 15 to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time they simply wanted to add or multiply a couple of numbers & assign the result to a variable. Such duplication of effort has 2 serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed.
2. it forces the programmer to program for the particular hardware & instruction set.

Structured programming. Main article: Wikipedia/structured programming. Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control & interfaces between components, with a reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points & (where possible) to have single exit points from functions & procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships & at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on & off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen & the database interface (which may itself be a standalone 3rd party library or a statically linked set of library routines).
- Either the database or the payroll application also has to initiate the process of exchanging data with between ship & shore, & that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of 1 component & its assorted internal methods from the others. Object-oriented programming embraces & extends this concept.

11.1.4 Data abstraction

Main article: Wikipedia/abstract data type. Data abstraction enforces a clear separation between the *abstract* properties of a data type & the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type – the *interface* to the data type – while the concrete implementation is kept entirely private, & indeed can change, e.g. to incorporate efficiency improvements over time. The idea is that such chances are not supposed to have any impact on client code, since they involve no difference in the abstract behavior.

E.g., one could define an abstract data type called *lookup table* which uniquely associates *keys* with values, & in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of `(key:value)` pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the 1st place, since any changes there can have major impacts on client code. As 1 way to look at this: the interface forms a *contract* on agreed behavior between the data type & client code; anything not spelled out in the contract is subject to change without notice.

11.1.5 Manual data abstraction

While much of data abstraction occurs through computer science & automation, there are times when this process is done manually & without programming intervention. 1 way this can be understood is through data abstraction within the process of conducting a systematic review of the literature. In this methodology, data is abstracted by 1 or several abstractors when conducting a meta-analysis, with errors reduced through dual data abstraction followed by independent checking, known as adjudication.

11.1.6 Abstraction in OOP

Main article: Wikipedia/object (computer science). In OOP theory, *abstraction* involves the facility to define objects that represent abstract "actors" that can perform work, report on & change their state, & "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, & standardizing the way that different data types interact, is the beginning of *abstraction*. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various OOP languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of 1 type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Commmon Lisp Object System or Self, e.g., feature less of a class-instance distinction & more use of delegation for polymorphism. Individual objects & functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates & overloading & other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter th need to support abstract nouns in code – all programming relies on an ability to abstract verbs as functions, nouns as data structures, & either as processes.

Consider e.g. a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger & feeding. It defines an `Animal` class to represent both the state of the animal & its functions:

```
public class Animal extends LivingThing
{
 private Location loc;
 private double energyReserves;
```

```
  public boolean isHungry() {
   return energyReserves < 2.5;
  }

  public void eat(Food food) {
   // Consume food
   energyReserves += food.getCalories();
  }

  public void moveTo(Location location) {
   // Move to new location
   this.loc = location;
  }
 }
```

With the above definition, one could create objects of type `Animal` & call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
 thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
 theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class `Animal` is an abstraction used in place of an actual animal, `LivingThing` is a further abstraction (in this case a generalization) of `Animal`.

If one requires a more differentiated hierarchy of animals – to differentiate, say, those who provide milk from those ho provide nothing except meat at the end of their lives – that is an intermediary level of abstraction, probably `DairyAnimal(cows,goats)` who would eat foods suitable to giving good milk, & `MeatAnimal(pigs,steers)` who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so they could concentrate instead on the feeding schedule. The 2 classes could be related using inheritance or stand alone, & the programmer could define varying degrees of polymorphism between the 2 types. These facilities tend to vary drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design. Main article: Wikipedia/object-oriented design. Decisions regarding what to abstract & what to keep under the control of the coder become the major concern of object-oriented design & domain analysis – actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time & budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs & cows & their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available & thus there is no reason to code the type of food into the class itself, & the design is a single simple `Animal` class of which pigs & cows are instances with the same functions. A decision to differentiate `DairyAnimal` would change the detailed analysis but the domain & legacy analysis would be unchanged – thus it is entirely under the control of the programmer, & it is called an abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

11.1.7 Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, *abstraction* refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. E.g., one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a *concrete* (more precise) model of execution.

Abstraction may be *exact* or *faithful* w.r.t. a property if one can answer a question about the property equally well on the concrete or abstract model. E.g., if one wishes to know what the result of the evaluation of a mathematical expression involving only integers $+, -, \cdot$, is worth module n, then one needs only perform all operations module n (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily *exact*, should be *sound*. I.e., it should be possible to get sound answers from them – even though the abstraction may simply yield a result of undecidability. E.g., students in a class may be abstracted by their minimal & maximal ages; if one asks whenever a certain person belongs to that class, one may simply compare that person's age with the minimal & maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction & other characteristics of the design, & how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because nontrivial properties of computer programs are essentially undecidable (Rice's theorem). As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

11.1.8 Levels of abstraction

Main article: Wikipedia/abstraction layer. Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information & processes, but with varying amounts of detail. Each level uses a system of expression involving a unique set of objects & compositions that apply only to a particular domain. Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. E.g., gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications & operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems. Main article: Wikipedia/database management system. Data abstraction levels of a database system. Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels"

- **Physical level.** The lowest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.
- **Logical level.** The next higher level of abstraction describes *what* data the database stores, & what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.
- **View level.** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide may views for the same database.

Layered architecture. Main article: Abstraction layer. The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction
- support the portability of software artifacts (model-based ideally)

System designs & business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). It is a technique used in designing computer software, hardware, & communications in which system or network components are isolated in layers so that changes can be made in 1 layer without affecting the others." – Wikipedia/abstraction (computer science)

11.2 Wikipedia/Anaconda

"*Anaconda* is an open source data science & AI distribution platform for Python & R programming languages. Developed by Anaconda, Inc., an American company founded in 2012, platform is used to develop & manage data science & AI projects. In 2024, Anaconda Inc. has about 300 employees & 45 million users.

11.2.1 History

Co-founded in Austin, Texas in 2012 as Continuum Analytics by *Peter Wang* & Travis Oliphant, Anaconda Inc. operates from US & Europe.

Anaconda Inc. developed Conda, a cross-platform, language-agnostic binary package manager. It also launched PyData community workshops & Jupyter Cloud Notebook service (Wakari.io). In 2013, it received funding from DARPA. In 2015, company had 2 million users including 200 of Fortune 500 companies & raised $24 million in a Series A funding round led by General Catalyst & BuildGroup. Anaconda secured an additional $30 million in funding in 2021.

Continuum Analytics rebranded as Anaconda in 2017. That year, it announced release of Anaconda Enterprise 5, an integration with Microsoft Azure, & had over 13 million users by year's end.

In 2022, it released Anaconda Business; new integrations with Snowflake & others; & open-source PyScript. It also required PythonAnywhere, while Anaconda's user base exceeded 30 million in 2022. In 2023, Anaconda released Python in Excel, a new integration with Microsoft Excel, & launched `PyScript.com`.

Company made a series of investments in AI during 2024. Feb, Anaconda partnered with IBM to import its repository of Python packages into Watsonx, IBM's generative AI platform. Same year, Anaconda joined IBM's AI Alliance & releasedan integration with Teradata & Lenovo.

In 2024, Anaconda's user base reached 45 million users & Barry Libert was named company CEO, after serving on Anaconda's board of directors.

11.2.2 Overview

Anaconda distribution comes with > 300 packages automatically installed, & > 7500 additional open-source packages can be installed from Anaconda repository as well as Conda package & virtual environment manager. It also includes a GUI, *Anaconda Navigator*, as a graphical alternative to command-line interface (CLI).

Conda was developed to address dependency conflicts native to pip package manager, which would automatically install any dependent Python packages without checking for conflicts with previously installed packages (until its version 20.3, which later implemented consistent dependency resolution). Conda package manager's historical differentiation analyzed & resolved these installation conflicts.

Anaconda is a distribution of Python & R programming languages for scientific computing (DS, ML applications, large-scale data processing, predictive analysis, etc.), that aims to simplify package management & deployment. Anaconda distribution includes data-science packages suitable for Windows, Linux, & macOS. Other company products include Anaconda Free, & subscription-based Starter, Business, & Enterprise. Anaconda's business tier offers Package Security Manager.

Package versions in Anaconda are managed by package management system Conda, which was spun out as a separate open-source package as useful both independently & for applications other than Python. There is also a small, bootstrap version of Anaconda called *Miniconda*, which includes only Conda, Python, packages they depend on, & a small number of other packages.

Open source packages can be individually installed from Anaconda repository, Anaconda Cloud (`anaconda.org`), or user's own private repository or mirror, using `conda install` command. Anaconda, Inc. compiles & builds packages available in Anaconda repository itself, & provides binaries for Windows 32/64 bit, Linux 64 bit & MacOS 64-bit (Intel, Apple Silicon). Anything available on PyPI may be installed into a Conda environment using `pip`, & Conda will keep track of what it has installed & what pip has installed. Custom packages can be made using `conda build` command, & can be shared with others by uploading them to Anaconda Cloud, PyPI or other repositories.

Default installation of Anaconda2 includes Python 2.7 & Anaconda3 includes Python 3.7. However, possible to create new environments that include any version of Python packaged with Conda.

- Anaconda Navigator. Anaconda Navigator is a desktop graphical user interface (GUI) included in Anaconda distribution that allows users to launch applications & manage Conda packages, environments & channels without using command-line commands. Navigator can search for packages on Anaconda Cloud or in a local Anaconda Repository, install them in an environment, run the packages & update them. Available for Windows, macOS & Linux.

  Applications are available by default in Navigator:

  - JupyterLab
  - Jupyter Notebook
  - QtConsole
  - Spyder
  - Glue
  - Orange
  - RStudio
  - Visual Studio Code

- Conda. Conda is an open source, cross-platform, language-agnostic package manager & environment management system that installs, runs, & updates packages & their dependencies. It was created for Python programs, but it can package & distribute software for any language (e.g., R), including multi-language projects. Conda package & environment manager is included in all versions of Anaconda, Miniconda, & Anaconda Repository.

11.2.3 Anaconda.org

Anaconda Cloud is a package management service by Anaconda where users can find, access, store & share public & private notebooks, environments, & Conda & PyPI packages. Cloud hosts useful Python packages, notebooks & environments for a wide variety of applications. Users do not need to log in or to have a Cloud account, to search for public packages, download & install them. Users can build new Conda packages using Conda-build & then use Anaconda Client CLI upload packages to `Anaconda.org`. Notebooks users can be aided with writing & debugging code with Anaconda's AI Assistant." – Wikipedia/Anaconda

11.3 Wikipedia/computational learning theory

"In CS, *computational learning theory* (or just *learning theory*) is a subfield of AI devoted to studying design & analysis of ML algorithms.

11.3.1 Overview

Theoretical results in ML mainly deal with a type of inductive learning called supervised learning. In supervised learning, an algorithm is given samples that are labeled in some useful way. E.g., samples might be descriptions of mushrooms, & labels could be whether or not mushrooms are edible. Algorithm takes these previously labeled samples & uses them to induce a classifier. This classifier is a function that assigns labels to samples, including samples that have not been seen previously by algorithm. Goal of supervised learning algorithm: optimize some measure of performance e.g. minimizing number of mistakes made on new samples.

In addition to performance bounds, computational learning theory studies time complexity & feasibility of learning. In computational learning theory, a computation is considered *feasible* (khả thi) if it can be done in polynomial time. There are 2 kinds of time complexity results:

- Positive results – showing: a certain class of functions is learnable in polynomial time.
- Negative results – showing: certain classes cannot be learned in polynomial time.

Negative results often rely on commonly believed, but yet unproven assumptions, e.g.:

- Computational complexity - P≠NP (P vs. NP problem)
- Cryptographic – 1-way functions exist.

There are several different approaches to computational learning theory based on making different assumptions about inference principles used to generalize from limited data. This includes different definitions of probability (see frequency probability, Bayesian probability) & different assumptions on generation of samples. Different approaches include:

- Exact learning, proposed by Dana Angluin
- Probably approximately correct learning (PAC learning), proposed by Leslie Valiant
- VC theory, proposed by Vladimir Vapnik & Alexey Chervonenkis
- Inductive inference as developed by Ray Solomonoff
- Algorithmic learning theory, from work of E. Mark Gold.
- Online ML, from work of Nick Littlestone.

While its primary goal is to understand learning abstractly, computational learning theory has led to development of practical algorithms. E.g., PAC theory inspired boosting, VC theory led to support vector machines, & Bayesian inference led to belief networks." – Wikipedia/computational learning theory

11.4 Wikipedia/Conda (package manager)

"*Conda* is an open-source, cross-platform, language-agnostic package manager & environment management system. It was originally developed to solve package management challenges faced by Python data scientists, & today is a popular package manager for Python & R. At 1st, Anaconda Python distribution was developed by Anaconda Inc.; later, it was spun out as a separate package, released under BSD license. Conda package & environment manager is included in all versions of Anaconda, Miniconda, & Anaconda Repository. Conda is a NumFOCUS affiliated project.

11.4.1 Features

As a package manger, Conda allows users to install different versions of binary software packages & their required software dependencies appropriate for their computing platform from a software repository. Conda checks everything that has been installed, any version limitations that user specifies (e.g., user wants a specific package to be at least version 2.1.3), & determines a set of versions for all requested packages & their dependencies that makes total set compatible with 1 another. If there is no set of compatible dependencies, it will tell user: requested combination of software packages at requested versions is not possible.

Secondly, Conda allows users to create such a set of software packages in isolation from rest of computing platform, in what Conda calls an *environment*. This allows user to create various sets of software packages for different projects. When users switches between those projects, they switch to relevant environment, thereby avoiding re-installation or removal of conflicting packages. To further facilitate setup of such environments, Conda can also install Python, interpreter for software packages itself.

Conda is written in Python programming language, but can manage projects containing code written in any language, including multi-language projects.

A popular Conda channel for bioinformatics software is *Bioconda*, which provides multiple software distributions for computational biology." – Wikipedia/Conda (package manager)

11.5 Wikipedia/data structure

A data structure known as a hash table. "In computer science, a *data structure* is a data organization & storage format that is usually chosen for efficient access to data. More precisely, a data structure is a collection of data values, the relationships among them, & the functions or operations that can be applied to the data, i.e., it is an algebraic structure about data.

11.5.1 Usage

Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

Different types of data structures are suited to different kinds of applications, & some are highly specialized to specific tasks. E.g., relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases & internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods & programming language emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage & retrieval (thu hồi) of information stored in both main memory & secondary memory.

11.5.2 Implementation

Data structures can be implemented using a variety of programming languages & techniques, but they all share the common goal of efficiently organizing & storing data. Data structures are generally based on the ability of a computer to fetch & store data at any place in its memory, specified by a pointer – a bit string, representing a memory address, that can be itself stored in memory & manipulated by the program. Thus, the array & record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself. This approach to data structuring has profound implications for the efficiency & scalability of algorithms. E.g., the contiguous memory allocation in arrays facilitates rapid access & modification operations, leading to optimized performance in sequential data processing scenarios.

The implementation of a data structure usually requires writing a set of procedures that create & manipulate instances of that structure. The efficiency of a data structure cannot ne analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, & the mathematical properties of those operations (including their space & time cost).

11.5.3 Examples

The standard type hierarchy of the programming language Python 3. Main article: Wikipedia/list of data structures. There are numerous types of data structures, generally built upon simpler primitive data types. Well known examples are:

- An array is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.

- A linked list (also just called *list*) is a linear collection of data elements of any type, called *nodes*, where each node has itself a value, & points to the next node in the linked list. The principal advantage of a linked list over an array is that values can always be efficiently inserted & removed without relocating the rest of the list. Certain other operations, e.g. random access to a certain element, are however slower on lists than on arrays.

- A record (also called *tuple* or *struct*) is an aggregate data structure. A record is a value that contains other values, typically in fixed number & sequence & typically indexed by names. The elements of records are usually called *fields* or *members*. In the context of OOP, records are known as plain old data structures to distinguish them from objects.

- Hash tables, also known as hash maps, are data structures that provide fast retrieval of values based on keys. They use a hashing function to map keys to indexes in an array, allowing for constant-time access in the average case. Hash tables are commonly used in dictionaries, caches, & database indexing. However, hash collisions can occur, which can impact their performance. Techniques like chaining & open addressing are employed to handle collisions.

- Graphs are collections of nodes connected by edges, representing relationships between entities. Graphs can be used to model social networks, computer networks, & transportation networks, among other things. They consist of vertices (nodes) & edges (connections between nodes). Graphs can be directed or undirected, & they can have cycles or be acylic. Graph traversal algorithms include breadth-1st search & depth-1st search.

- Stacks & queues are abstract data types that can be implemented using arrays or linked lists. A stack has 2 primary operations: push (adds an element to the top of the stack) & pop (removes the topmost element from the stack), that follow the Last In, First Out (LIFO) principle. Queues have 2 main operations: enqueue (adds an element to the rear of the queue) & dequeue (removes an element from the front of the queue) that follow the First In, First Out (FIFO) principle.

- Trees represents a hierarchical organization of elements. A tree consists of nodes connected by edges, with 1 node being the root & all other nodes forming subtrees. Trees are widely used in various algorithms & data storage scenarios. Binary trees (particularly heaps), AVL trees, & B-trees are some popular types of trees. They enable efficient & optimal searching, sorting, & hierarchical representation of data.

A trie, or prefix tree, is a special type of tree used to efficiently retrieve strings. In a trie, each node represents a character of a string, & the edges between nodes represent the characters that connect them. This structure is especially useful for tasks like autocomplete, spell-checking, & creating dictionaries. Tries allow for quick searches & operations based on string prefixes.

11.5.4 Language support

Most assembly languages & some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many high-level programming languages & some higher-level assembly languages, e.g. MASM, have special syntax or other built-in support for certain data structures, e.g. records & arrays. E.g., the C (a direct descendant of BCPL) & Pascal languages support structs & records, respectively, in addition to vectors (1D arrays) & multi-dimensional arrays.

Most programming languages feature some sort of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. E.g., C++ Standard Template Library, Java Collections Framework, & Microsoft .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module & its implementation. Some provide apaque data types that allow clients to hide implementation details. OOP languages, e.g. C++, Java, & Smalltalk, typically use classes for this purpose.

Many known data structures have concurrent versions which allow multiple computing threads to access a single concrete instance of a data structure simultaneously." – Wikipedia/data structure

11.6 Wikipedia/level set (data structures)

"In computer science, a level set data structure is designed to represent discretely sampled dynamic level sets functions. A common use of this form of data structure is in efficient image rendering. The underlying method constructs a signed distance field that extends the boundary, & can be used to solve the motion of the boundary in this field.

11.6.1 Chronological developments

The powerful level-set method is due to Osher & Sethian 1988. However, the straightforward implementation via a dense d-dimensional array of values, results in both time & storage complexity of $O(n^d)$, where n is the cross sectional resolution of the spatial extents of the domain & d is the number of spatial dimensions of the domain.

1. **Narrow band.** The narrow band level set method, introduced in 1995 by Adalsteinsson & Sethian, restricted most computations to a thin band of active voxels immediately surrounding the interface, thus reducing the time complexity in 3D to $O(n^2)$ for most operations. Periodic updates of the narrowband structure, to rebuild the list of active voxels, were required which entailed an $O(n^3)$ operation in which voxels over the entire volume were accessed. The storage complexity for this narrowband scheme was still $O(n^3)$. Differential constructions over the narrow band domain edge require careful interpolation & domain alternation schemes to stabilize the solution.

2. **Sparse field.** This $O(n^3)$ time complexity was eliminated in the approximate "sparse field" level set method introduced by Whitaker in 1998. The sparse field level set method employs a set of linked list to track the active voxels around the interface. This allows incremental extension of the active region as needed without incurring any significant overhead. While consistently $O(n^2)$ efficient in time, $O(n^3)$ storage space is still required by the sparse field level set method.

3. **Sparse block grid.** The sparse block grid method, introduced by Bridson in 2003, divides the entire bounding volume of size n^3 into small cubic blocks of m^3 voxels each. A coarse grid of size $\left(\frac{n}{m}\right)^3$ then stores pointers only to those blocks that intersect that narrow band of the level set. Block allocation & deallocation occur as the surface propagates to accommodate to the deformations. This method has a suboptimal storage complexity of $O(\left(\frac{n}{m}\right)^3 + m^3n^2)$, but retains the constant time access inherent to dense grids.

4. **Octree.** The octree level set method, introduced by Strain in 1999 & refined by Losasso, Gibu, & Fedkiw, & more recently by Min & Gibou uses a tree of nested cubes of which the leaf nodes contain signed distance values. Octree level sets currently require uniform refinement along the interface (i.e., the narrow band) in order to obtain sufficient precision. This representation is efficient in terms of storage, $O(n^2)$, & relatively efficient in terms of access queries, $O(\log n)$. A advantage of the level method on octree data structures is that one can solve the PDEs associated with typical free boundary problems that use the level set method. The CASL research group has developed this line of work in computational materials, CFD, electrokinetics, image guided surgery & controls.

5. **Run-length encoded.** The run-length encoding (RLE) level set method, introduced in 2004, applies the RLE scheme to compress regions away from the narrow band to just their sign representation while storing with full precision the narrow band. The sequential traversal of the narrow band is optimal & storage efficiency is further improved over the octree level set. The additional of an acceleration lookup table allows for fast $O(\log r)$ random access, where r is the number of runs per cross section. Additional efficiency is gained by applying the RLE scheme in a dimensional recursive fashion, a technique introduced by Nielsen & Museth's similar DT-Grid.

6. **Hash Table Local Level Set.** The Hash Table Local Level Set method, introduced in 2011 by Eyiyurekli & Breen & extended in 2012 by Brun, Guittet, & Gibou, only computes the level set data in a band around the interface, as in the Narrow Band Level-Set Method, but also only stores the data in that same band. A hash table data structure is used, which provides an $O(1)$ access to the data. However, Brun et al. conclude that their method, while being easier to implement, performs worse than a quadtree implementation. They find that

   > as it is, [. . .] a quadtree data structure seems more adapted than the hash table data structure for level-set algorithms.

   3 main reasons for worse efficiency are listed:

   (a) to obtain accurate results, a rather large band is required close to the interface, which counterbalances the absence of grid nodes far from the interface;

   (b) the performances are deteriorated by extrapolation procedures on the outer edges of the local grid &

   (c) the width of the band restricts the time step & slows down the method.

   (d) **Point-based.** Corbett in 2005 introduced the point-based level set method. Instead of using a uniform sampling of the level set, the continuous level set function is reconstructed from a set of unorganized point samples via moving least squares." – Wikipedia/level set (data structures)

11.7 Wikipedia/pip (package manager)

"`pip` (also known by Python 3's alias `pip3`) is a package-management system written in Python & is used to install & mange software packages. Python Software Foundation recommends using pip for installing Python applications & its dependencies during deployment. Pip connects to an online repository of public packages, called Python Package Index. Pip can be configured to connect to other package repositories (local or remote), provided that they comply to Python Enhancement Proposal 503.

Most distributions of Python come with pip preinstalled. Python 2.7.9 & later (on python2 series), & Python 3.4 & later include pip by default.

11.7.1 History

1st introduced as `pyinstall` in 2008 by Ian Bicking (creator of `virtualenv` package) as an alternative to easy install, pip was chosen as new name from 1 of several suggestions: creator received on his blog post. According to Bicking himself, the name is a recursive acronym for "Pip Installs Packages". In 2011, Python Packaging Authority (PyPA) was created to take over maintenance of pip & virtualenv from Bicking, led by Carl Meyer, Brian Rosner, & Jannis Leidel.

With release of pip version 6.0 (2014-12-22), the version naming process was changed to have version in X.Y format & drop preceding 1 from version label.

11.7.2 Command-line interface

An output of `pip install virtualenv`. Pip's command-line interface allows install of Python software packages by issuing a command: `pip install some-package-name`. Users can also remove package by issuing a command: `pip uninstall some-package-name`. pip has a feature to manage full lists of packages & corresponding version numbers, possible through a "requirements" file. This permits efficient re-creation of an entire group of packages in a separate environment (e.g. another computer) or virtual environment. This can be achieved with a properly formatted file & following command, where `requirements.txt` is name of file: `pip install -r requirements.txt`.

To install some package for a specific python version, pip provides following command, where `${version}` is replaced by 2, 3, 3.4, etc.: `pip${version} install some-package-name`.

- Using `setup.py`. Pip provides a way to install user-defined projects locally with use of `setup.py` file. This method requires python project to have following file structure.

  Within this structure, user can add `setup.py` to root of project (i.e. `example_project` for above structure) with following content:

```
from setuptools import setup, find_packages

setup(
    name='example',  # Name of the package. This will be used, when the project is imported as a package.
    version='0.1.0',
    packages=find_packages(include=['exampleproject', 'exampleproject.*'])  # Pip will automatically install
)
```

  After this, pip can install this custom project by running following command, from project root directory: `pip install -e`.

11.7.3 Custom repository

Besides default PyPI repository, Pip supports custom repositories as well. Such repositories can be located on an HTTP(s) URL or on a file system location. A custom repository can be specified using the `-i` or `-index-url` option, like so:

```
pip install -i https://your-custom-repo/simple <package name>
```

or with a filesystem: `pip install -i /path/to/your/custom-repo/simple <package name>`." – Wikipedia/pip (package manager)

Tài liệu

[Bos14] Wes Bos. *Sublime Text Power User: A Complete Guide*. 2014, p. 202.

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

[CS14] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Apress, 2014, p. 458.

[Dok20] Jørgen S. Dokken. "Automatic shape derivatives for transient PDEs in FEniCS and Firedrake". In: (2020). URL: https://arxiv.org/abs/2001.10058.

[DT06] Lê Văn Doanh and Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal*. In lần thứ 10. Nhà Xuất Bản Khoa Học & Kỹ Thuật, 2006, p. 268.

[Đức22] Nguyễn Tiến Đức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python*. Nhà Xuất Bản Đại Học Thái Nguyên, 2022, p. 327.

[DV21] Christoph Dürr and Jill-Jênn Vie. *Competitive Programming in Python: 128 Algorithms to Develop Your Coding Skills*. Undergraduate Topics in Computer Science. Translated by Greg Gibbons & Danièle Gibbons. Cambridge University Press, 2021, pp. x+254.

[GR09] Christophe Geuzaine and Jean-François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *Internat. J. Numer. Methods Engrg.* 79.11 (2009), pp. 1309–1331. ISSN: 0029-5981. DOI: 10.1002/nme.2579. URL: https://doi.org/10.1002/nme.2579.

[GW22] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022.

[Huy24] Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1*. Tái bản lần 10. Nhà Xuất Bản Thông Tin & Truyền Thông, 2024, p. 371.

[Kho06] Nguyễn Văn Khoa. *Nghệ Thuật Lập Trình Máy Tính*. Nhà Xuất Bản Giao Thông Vận Tải, 2006, p. 1050.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. 3rd edition. Addison-Wesley Professional, 1997, pp. xx+652.

[Laa20] Antti Laaksonen. *Guide to Competitive Programming: Learning & Improving Algorithms Through Contests*. 2nd edition. Undergraduate Topics in Computer Science. Springer, 2020, pp. xv+309.

[Laa24] Antti Laaksonen. *Guide to Competitive Programming: Learning & Improving Algorithms Through Contests*. 3rd edition. Undergraduate Topics in Computer Science. Springer, 2024, pp. xviii+349.

[LL16] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Vol. 3. Simula SpringerBriefs on Computing. The FEniCS tutorial I. Springer, Cham, 2016, pp. ix+148. ISBN: 978-3-319-52461-0; 978-3-319-52462-7. DOI: 10.1007/978-3-319-52462-7. URL: https://doi.org/10.1007/978-3-319-52462-7.

[Ngọ02] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C*. Nhà Xuất Bản Thống Kê, 2002, p. 425.

[Ngọ03] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++*. Nhà Xuất Bản Thống Kê, 2003, p. 476.

[Ngọ08] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Thống Kê, 2008, p. 338.

[Ngọ09] Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Giáo Dục, 2009, p. 187.

[Pel13] Dan Peleg. *Mastering Sublime Text*. Packt Publishing, Birmingham - Mumbai, 2013, pp. iv+94.

[PW20] Alberto Paganini and Florian Wechsung. "Fireshape Documentation, Release 0.0.1". In: (2020), pp. ii+31. URL: https://fireshape.readthedocs.io/en/latest/index.html.

[PW21] Alberto Paganini and Florian Wechsung. "Fireshape: a shape optimization toolbox for Firedrake". In: *Struct. Multidiscip. Optim.* 63.5 (2021), pp. 2553–2569. ISSN: 1615-147X. DOI: 10.1007/s00158-020-02813-y. URL: https://doi.org/10.1007/s00158-020-02813-y.

[Sho19] William Shotts. "The Linux Command Line: A Complete Introduction". In: (2019), p. 640.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th edition. Pearson Addison-Wesley, 2013, pp. xiv+1346.

[Str18] Bjarne Stroustrup. *A Tour of C++*. 2nd edition. Pearson Addison-Wesley, 2018, pp. xii+240.

[TN13] M. Towara and U. Naumann. "A Discrete Adjoint Model for OpenFOAM". In: *Procedia Computer Science* 18 (2013), pp. 429–438. DOI: 10.1016/j.procs.2013.05.206. URL: https://doi.org/10.1016/j.procs.2013.05.206.

[Wil+17] Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, and et al. "ParMooN—a modernized program package based on mapped finite elements". In: *Comput. Math. Appl.* 74.1 (2017), pp. 74–88. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2016.12.020. URL: https://doi.org/10.1016/j.camwa.2016.12.020.

[WW16] Yonghui Wu and Jiande Wang. *Data Structure Practice for Collegiate Programming Contests & Education*. 1st edition. CRC Press, 2016, p. 496.

[WW18] Yonghui Wu and Jiande Wang. *Algorithm Design Practice for Collegiate Programming Contests & Education*. 1st edition. CRC Press, 2018, p. 692.