How to Design Algorithms and Data Structures in Practice

How to Design Algorithms and Data Structures

Really understand the problem –

- specifications
  - what exactly does the input consist of?
  - what exactly are the desired results / output?
  - construct a small input example – what happens when you try to solve it by hand?
  - is the problem sufficiently well defined to actually have a correct solution?
- assess the requirements
  - how large is a typical instance?
  - is it necessary to find the optimal solution? is close good enough?
  - how important is speed? how long is it acceptable to wait?
  - how much time and effort do you have to spend?
- identify a strategy
  - what kind of problem is it? (numerical, graph, geometric, string, set, …) what kind of formulation seems easiest?

How to Design Algorithms and Data Structures

Try a simple solution –

- basic properties
  - how do you measure the quality of a solution once constructed?
- brute force – straightforward implementation of definition, search through all possible solutions and pick the best
  - will this work correctly? if so, why are you sure of that?
  - is the running time polynomial or exponential? is the typical input size small enough that it doesn't matter?
- heuristic – repeatedly apply a simple rule about what to do next
  - for what kinds of inputs does this strategy work well enough? do you need to solve the problem for other inputs?
  - for what kinds of inputs does this strategy work poorly? if you can't find any examples, can you prove that it always works well?
  - how quickly does this strategy find an answer? is the implementation simple?

How to Design Algorithms and Data Structures

In a simple solution is insufficient, start trying on hats –

- can you recognize the problem as something familiar?
  - identify the essence of the problem
  - consult the Hitchhiker's Guide / Stony Brook Algorithm Repository (or another source) Skiena, *The Algorithm Design Manual* http://www.algorist.com/algorist.html
  - is the problem a special case of something familiar?
- if so, what is known about the problem? is there an implementation that you can use?
- if not, did you look in the right place?
  - browse the Hitchhiker's Guide more carefully
  - look under all possible keywords in the index
  - search other algorithms resources and the Internet

How to Design Algorithms and Data Structures

If the problem isn't recognizable as something familiar –

- consider special cases to gain insight
  - can you simplify the problem enough to solve it efficiently? e.g. ignore some parameters, set some parameters to trivial values, ignore some aspects of the task
  - why can't the special-case solution be generalized?

How to Design Algorithms and Data Structures

- design a solution
  - is there something that can be sorted? does that make it easier to find the answer?
  - is there a way to split the problem into two smaller problems? can divide-and-conquer be used?
  - do the input elements or solution have a natural left-to-right order? can the problem be formulated as a series of decisions? can dynamic programming be used to exploit this order?
  - are certain operations done repeatedly, such as searching or finding max/min? can a data structure (map, PQ) be used to speed this up?
  - does the problem sound like an NP-complete problem?
    - consult a list of NP-complete problems
    - try tactics for dealing with NP-complete problems

How to Design Algorithms and Data Structures

Other strategies –

- consider randomness
  - e.g. randomly choosing the next item to consider
  - e.g. random sampling
  - e.g. simulated annealing
- can the problem be formulated as a linear program? an integer program?

Course Takeaways

- knowledge of how to think about algorithms and data structures
  - developing an efficient data structure for a problem, based on an analysis of the problem (including adapting typical ADTs/data structures as needed)
  - developing an efficient and correct algorithm for a problem, including any necessary data structures
  - justifying decisions made, demonstrating a thorough consideration of the implications of the choices made, tradeoffs, and alternatives that were dismissed

Course Takeaways

- a working knowledge of algorithmic efficiency
  - determining the time and space requirements of data structures and algorithms (both iterative and recursive)
  - having a sense whether the time and space requirements are good or whether improvements seem likely (and where to look for them)

Course Takeaways

- a toolbox of ADTs, data structures, and algorithmic strategies
  - know the characteristic operations of the ADTs studied, and be able to identify ADT(s) appropriate for a given application
  - know the time and space requirements of typical operations in the data structures studied, and be able to select an appropriate implementation for a given application
  - know what characteristics make a problem suitable for a particular algorithmic technique (and be able to recognize when a problem is not suitable for a particular technique)
  - know the "templates" or patterns for applying the algorithmic techniques studied to develop an algorithm and prove it correct

Data Structures

Dictionaries, Priority Queues, Suffix Trees and Arrays, Graph Data Structures, Set Data Structures, Kd-Trees

Numerical Problems

Solving Linear Equations, Bandwidth Reduction, Matrix Multiplication, Determinants and Permanents, Constrained and Unconstrained Optimization, Linear Programming, Random Number Generation, Factoring and Primality Testing, Arbitrary-Precision Arithmetic, Knapsack Problem, Discrete Fourier Transform

Combinatorial Problems

Sorting, Searching, Median and Selection, Generating Permutations, Generating Subsets, Generating Partitions, Generating Graphs, Calendrical Calculations, Job Scheduling, Satisfiability

Graph: Polynomial-time Problems

Connected Components, Topological Sorting, Minimum Spanning Tree, Shortest Path, Transitive Closure and Reduction, Matching, Eulerian Cycle/Chinese Postman, Edge and Vertex Connectivity, Network Flow, Drawing Graphs Nicely, Drawing Trees, Planarity Detection and Embedding

Graph: Hard Problems

Clique, Independent Set, Vertex Cover, Traveling Salesman Problem, Hamiltonian Cycle, Graph Partition, Vertex Coloring, Edge Coloring, Graph Isomorphism, Steiner Tree, Feedback Edge/Vertex Set

Computational Geometry

Robust Geometric Primitives, Convex Hull, Triangulation, Voronoi Diagrams, Nearest Neighbor Search, Range Search, Point Location, Intersection Detection, Bin Packing, Medial-Axis Transform, Polygon Partitioning, Simplifying Polygons, Shape Similarity, Motion Planning, Maintaining Line Arrangements, Minkowski Sum

Set and String Problems

Set Cover, Set Packing, String Matching, Approximate String Matching, Text Compression, Cryptography, Finite State Machine Minimization, Longest Common Substring/Subsequence, Shortest Common Superstring

https://www.algorist.com/algorist.html