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

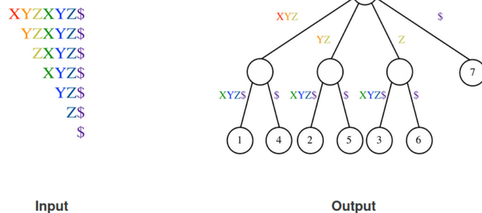
Set and String Problems

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

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

Suffix Trees and Arrays



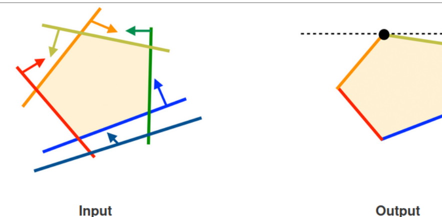
Input Description: A reference string S.
Problem: Build a data structure for quickly finding all places where an arbitrary query string q is a substring of S.

Excerpt from The Algorithm Design Manual: Suffix trees and arrays are phenomenally useful data structures for solving string problems elegantly and efficiently. If you need to speed up a string processing algorithm from $O(n^2)$ to linear time, proper use of suffix trees is quite likely the answer.

Linear Programming

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

Maximizing $6x_1 + x_2 + 4x_3$
subject to
$3x_1 + 7x_2 + x_3 \leq 15$
$x_1 - 2x_2 + 3x_3 \leq 20$
$x_1, x_2, x_3 \geq 0$



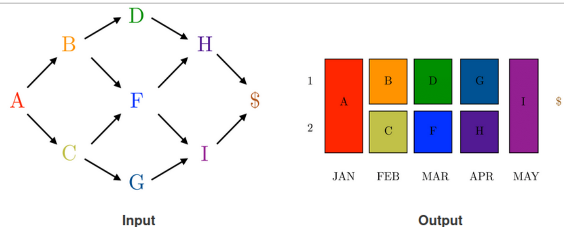
Input Description: A set of linear inequalities, a linear objective function.
Problem: Find the assignment to the variables maximizing the objective function while satisfying all inequalities.

Excerpt from The Algorithm Design Manual: The standard algorithm for linear programming is called the simplex method. Each constraint in a linear programming problem acts like a knife that carves away a region from the space of possible solutions. We seek the point within the remaining region that maximizes (or minimizes) $f(X)$. By appropriately rotating the solution space, the optimal point can always be made to be the highest point in the region. Since the region (simplex) formed by the intersection of a set of linear constraints is convex, we can find the highest point by starting from any vertex of the region and walking to a higher neighboring vertex. When there is no higher neighbor, we are at the highest point.

While the basic simplex algorithm is not too difficult to program, there is a considerable art to producing an efficient implementation capable of solving large linear programs. For example, large programs tend to be sparse (meaning that most inequalities use few variables), so sophisticated data structures must be used. There are issues of numerical stability and robustness, as well as which neighbor we should walk to next (so called *pivoting rules*). Finally, there exist sophisticated *interior-point* methods, which cut through the interior of the simplex instead of walking along the outside, that beat simplex in many applications.

Job Scheduling

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



Input Description: A directed acyclic graph $G = (V, E)$, where the vertices represent jobs and the the edge (u, v) that task u must be completed before task v.
Problem: What schedule of tasks to completes the job using the minimum amount of time or processors?

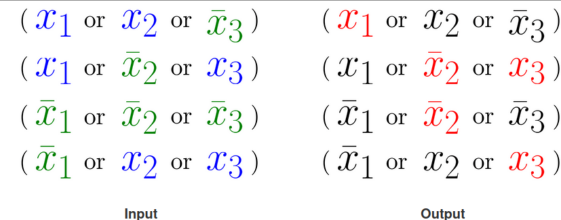
Excerpt from The Algorithm Design Manual: Devising a proper schedule to satisfy a set of constraints is fundamental to many applications. A critical aspect of any parallel processing system is the algorithm mapping tasks to processors. Poor scheduling can leave most of the expensive machine sitting idle while one bottleneck task is performed. Assigning people to jobs, meetings to rooms, or courses to final exam periods are all different examples of scheduling problems.

Scheduling problems differ widely in the nature of the constraints that must be satisfied and the type of schedule desired. For this reason, several other catalog problems have a direct application to various kinds of scheduling.

We focus on precedence-constrained scheduling problems for directed acyclic graphs. These problems are often called PERT/CPM, for *Program Evaluation and Review Technique/Critical Path Method.* Suppose you have broken a big job into a large number of smaller tasks. For each task you know how long it should take (or perhaps an upper bound on how long it might take). Further, for each pair of tasks you know whether it is essential that one task be performed before another. The fewer constraints we have to enforce, the better our schedule can be. These constraints must define a directed acyclic graph, acyclic because a cycle in the precedence constraints represents a Catch-22 situation that can never be resolved.

Satisfiability

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



Input Description: A set of clauses in conjunctive normal form.
Problem: Is there a truth assignment to the boolean variables such that every clause is satisfied?

Excerpt from The Algorithm Design Manual: Satisfiability arises whenever we seek a configuration or object that must be consistent with (\ie satisfy) a given set of constraints. For example, consider the problem of drawing name labels for cities on a map. For the labels to be legible, we do not want the labels to overlap, but in a densely populated region many labels need to be drawn in a small space. How can we avoid collisions?

For each of the n cities, suppose we identify two possible places to position its label, say right above or right below each city. We can represent this choice by a Boolean variable \(v_i\), which will be true if city \(c_i\)'s label is above \(c_i\), otherwise \(v_i = false\). Certain pairs of labels may be forbidden, such as when \(c_i\)'s above label would obscure \(c_j\)'s below label. This pairing can be forbidden by the two-element clause $(\bar{v}_i \backslash \mathrm{OR} v_j)$, where $\bar{v}$ means not v". Finding a satisfying truth assignment for the resulting set of clauses yields a mutually legible map labeling if one exists.

Satisfiability is *the* original NP-complete problem. Despite its applications to constraint satisfaction, logic, and automatic theorem proving, it is perhaps most important theoretically as the root problem from which all other NP-completeness proofs originate.

Graph: Polynomial-time Problems

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

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

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

Clique

Independent Set

Vertex Cover

Traveling Salesman Problem

Hamiltonian Cycle

Graph Partition

Vertex Coloring

Edge Coloring

Graph Isomorphism

Steiner Tree

Feedback Edge/Vertex Set

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

Bin Packing

A B C D E F G H

Input

H C D E A F B G

Output

Input Description: A set of n items with sizes $d_1, \ldots, d_n$. A set of m bins with capacity $c_1, \ldots, c_m$.
Problem: How do you store the set of items using the fewest number of bins?

Excerpt from The Algorithm Design Manual: Bin packing arises in a variety of packaging and manufacturing problems. Suppose that you are manufacturing widgets with parts cut from sheet metal, or pants with parts cut from cloth. To minimize cost and waste, we seek to lay out the parts so as to use as few fixed-size metal sheets or bolts of cloth as possible. Identifying which part goes on which sheet in which location is a bin-packing variant called the *cutting stock* problem. After our widgets have been successfully manufactured, we will be faced with another bin packing problem, namely how best to fit the boxes into trucks to minimize the number of trucks needed to ship everything.

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

Convex Hull

Input

Output

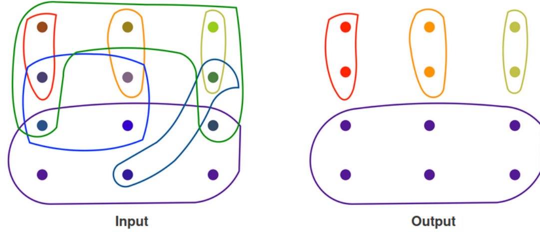
Input Description: A set S of n points in d-dimensional space.
Problem: Find the smallest convex polygon containing all the points of S.

Excerpt from The Algorithm Design Manual: Finding the convex hull of a set of points is *the* most elementary interesting problem in computational geometry, just as minimum spanning tree is the most elementary interesting problem in graph algorithms. It arises because the hull quickly captures a rough idea of the shape or extent of a data set.

Convex hull also serves as a first preprocessing step to many, if not most, geometric algorithms. For example, consider the problem of finding the diameter of a set of points, which is the pair of points a maximum distance apart. The diameter will always be the distance between two points on the convex hull. The *O(n \lg n)*. algorithm for computing diameter proceeds by first constructing the convex hull, then for each hull vertex finding which other hull vertex is farthest away from it. This so-called rotating-calipers" method can be used to move efficiently from one hull vertex to another.

Set Packing

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



Input Description: A set of subsets $S = S_1, \ldots, S_m$ of the universal set $U = \{1, \ldots, n\}$.

Problem: What is the largest number of mutually disjoint subsets from S?

Excerpt from The Algorithm Design Manual: Set packing problems arise in partitioning applications, where we need to partition elements under strong constraints on what is an allowable partition. The key feature of packing problems is that no elements are permitted to be covered by more than one set. We seek a large subset of vertices such that each edge is adjacent to at most one of the selected vertices. To model this as set packing, let the universal set consist of all edges of G, and subset \\(S_i\\) consist of all edges incident on vertex \\(v_i\\). Any set packing corresponds to a set of vertices with no edge in common, in other words, an independent set.

Scheduling airline flight crews to airplanes is another application of set packing. Each airplane in the fleet needs to have a crew assigned to it, consisting of a pilot, copilot, and navigator. There are constraints on the composition of possible crews, based on their training to fly different types of aircraft, as well as any personality conflicts. Given all possible crew and plane combinations, each represented by a subset of items, we need an assignment such that each plane and each person is in exactly one chosen combination. After all, the same person cannot be on two different planes, and every plane needs a crew. We need a perfect packing given the subset constraints.