Algorithm Concepts

David R. Musser Brian Osman

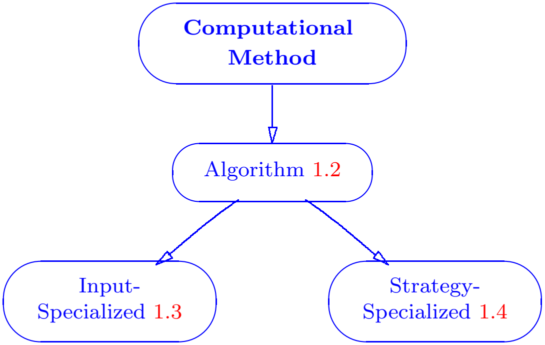
May 16, 2003

Click here for printer-friendly version

This document contains Section 1 of *Algorithm Concepts*, a collection of algorithm concept descriptions in both Web page and print form under development at Rensselaer Polytechnic Institute by David R. Musser, with the aid of graduate research assistants Brian Osman, Michael LaSpina, and Mayuresh Kulkarni, and with significant participation also of students in the "Adopt an Algorithm" project in CSCI-4020 Computer Algorithms, Spring 2002 and Spring 2003.

1. Basic Algorithm Concepts

1.1. Computational Method



A *computational method* is a method for solving a specific type of problem by means of a finite set of steps operating on *inputs*, which are quantities given to it before execution of the steps begins or during executing, and producing one or more *outputs*, which have a specified relation to the inputs. The number of steps in the method is required to be not only finite but also independent of the inputs. (The program does not grow or shrink in response to the inputs, but it might have different variations for different types of inputs.) The method is also required to be *resource constrained*, which means there are requirements on all operations of all steps of the method that constrain the resources (time, space) that can used in executions of the method.

Execution of steps may repeat other steps, so that although the set of steps is finite, executions of them may produce an infinite sequence of steps—finite termination is not a requirement (it *is* a requirement of the algorithm (§1.2) concept). Some nonterminating computational methods are useful, such as computer operating systems or event-driven simulation systems. Even though the execution of such methods does not terminate, we are still generally interested in bounding the number of steps taken in producing some partial output (as in proving response-time guarantees for an operating system).

In order to bound the resources—time and space—consumed during an execution of the method, we first need bounds on the resources consumed by individual steps. This motivates the resource-constraint requirement on computational methods.

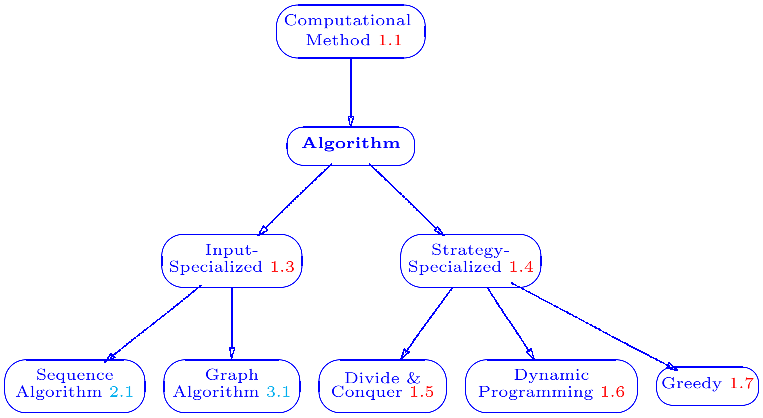
Effectiveness of a computational method is the property that all operations of all steps of the method "must be sufficiently basic that they can in principle be done exactly and in a finite length of time." (Knuth [1] adds "by someone using pencil and paper," but it is a philosophical question whether humans have any computational capability beyond the effectiveness of machines.) As defined here, effectiveness of computational methods follows from their resource-constraint requirement.

Definiteness of a computational method is the property that each step of the method "must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case" [1]. This includes the property that it must be unambiguous which step, if any, follows the current step in any execution of the method.

Again, resource-constraint requirements place some limitations on just how "indefinite" the steps of a method may be.

Refinements: Algorithm (§1.2)

1.2. Algorithm



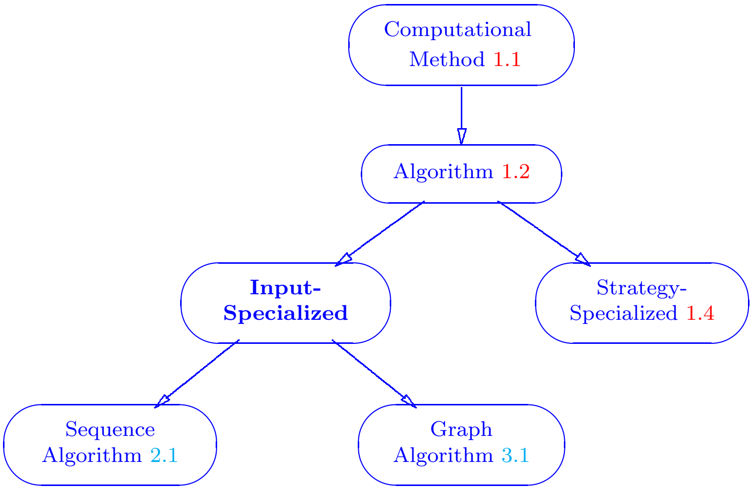
Refinement of: Computational Method (§1.1)

Finiteness of a computational method is the property that the number of steps in any execution of the method must be finite. The finiteness property is also called *termination*, and the method is said to be *terminating*. *Algorithm* is a synonym for *finite computational method*, a computational method (§1.1) with the additional property of finiteness. Every abstraction that belongs to an algorithm concept must have the termination property.

Note that among the abstractions belonging to a computational method concept, some might be terminating while others are nonterminating.

Refinements: Algorithm Specialized by Input (§1.3), Algorithm Specialized by Strategy (§1.4).

1.3. Algorithm Specialized by Input

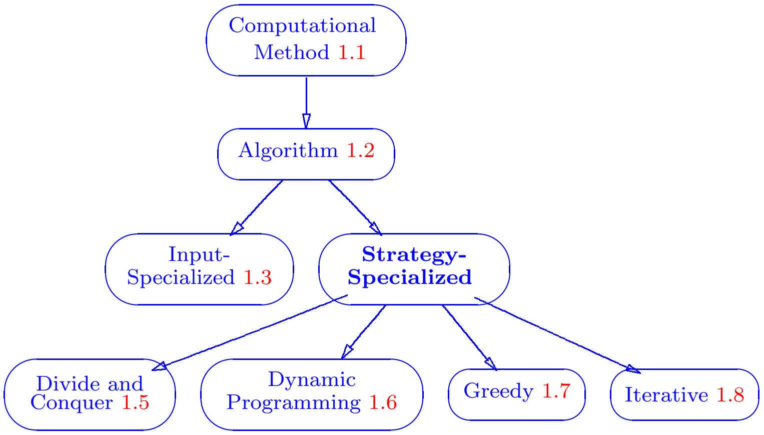


Refinement of: Algorithm (§1.2)

This concept is a narrowing of the algorithm (§1.2) concept by restrictions on the form of input. Subconcepts restrict their input to some particular domain, such as sets, graphs, or linear sequences.

Refinements: Set Algorithm, Sequence Algorithm (§2.1), Polynomial Algorithm, Matrix Algorithm, Graph Algorithm (§3.1)

1.4. Algorithm Specialized by Strategy

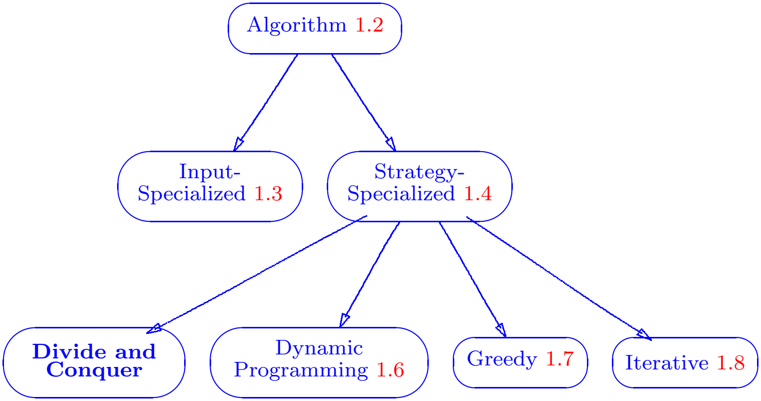


Refinement of: Algorithm (§1.2)

This concept is a narrowing of the algorithm (§1.2) concept in terms of strategies used in structuring the steps of the algorithm.

Refinements: Divide-and-Conquer Algorithm (§1.5), Dynamic Programming Algorithm (§1.6), Greedy Algorithm (§1.7), Iterative Algorithm (§1.8).

1.5. Divide-and-Conquer Algorithm



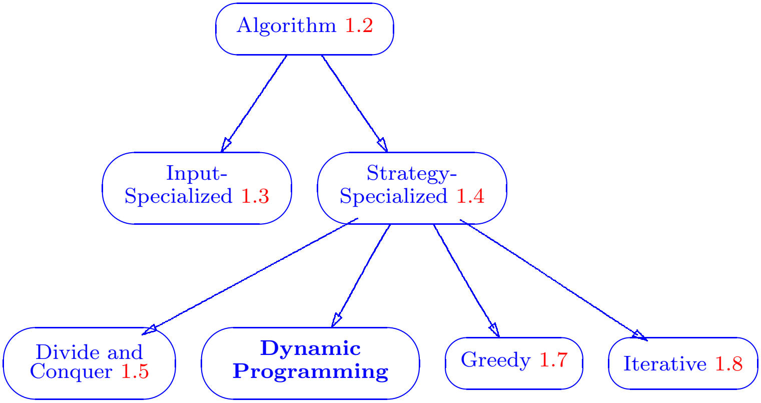
A *divide-and-conquer algorithm* is an algorithm (§1.2) whose steps are structured according to the following strategy:

1. Construct the output directly and return it, if the input is simple enough. Otherwise:
2. Divide the input into two or more (a finite number) of smaller inputs.
3. Recursively apply the algorithm to each of the smaller inputs produced in the first step.

4. Combine the outputs from the recursive applications to produce the output corresponding to the original input.

This concept is one of many known ways of narrowing the algorithm concept in terms of a strategy (§1.4), which gives a specific structure to the steps of the algorithm.

1.6. Dynamic Programming Algorithm



Refinement of: Algorithm Specialized by Strategy (§1.4)

A *dynamic programming algorithm* is an algorithm which solves a given problem by combining solutions to smaller subproblems. The strategy depends on two characteristics of the problem to be solved, optimal substructure and overlapping subproblems.

Optimal substructure: A problem is said to have optimal substructure if the optimal solution to the problem contains within it optimal solutions to the contained subproblems.

Overlapping subproblems: A problem exhibits overlapping subproblems if the total number of subproblems required to assemble and solve the complete problem is "small," generally polynomial in the input size. In other words, a naive recursive (top down) approach to the problem would recompute the solution to the subproblems many times.

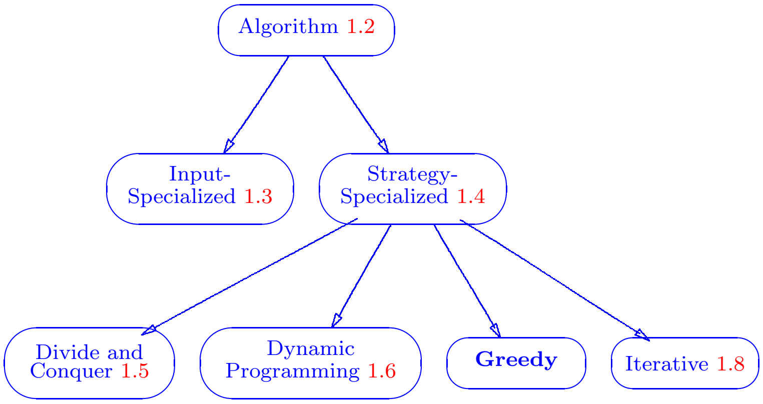
Taking advantage of the above properties, a dynamic programming algorithm functions in a bottom up fashion. The overall strategy can be descibed as:

1. Compute and store the solutions to all of the simplest subproblems.
2. Repeat until the full problem has been solved:
   (a) Combine the solutions to the subproblems of a given size to compute and store the solutions to the next largest subproblems.

As can be seen, the solutions to the various subproblems are stored for repeated access in computing the solutions to larger subproblems. This storage is often done in some table, and dynamic programming is sometimes referred to as a *tabular method*.

This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy (§1.4), which gives a specific strategy to the steps of the algorithm.

1.7. Greedy Algorithm



Refinement of: Algorithm Specialized by Strategy (§1.4)

A *greedy algorithm* is an algorithm which always makes locally optimal choices during its execution to produce a globally optimal solution to some problem. For such a strategy to work, the problem must exhibit the greedy choice property, and optimal substructure.

Greedy choice property: A problem exhibits the greedy choice property if a globally optimal solution can be arrived at by making locally optimal decisions at every decision point. In other words, the subproblems which would result from various decisions, and their resulting solutions to the whole problem, are irrelevant.

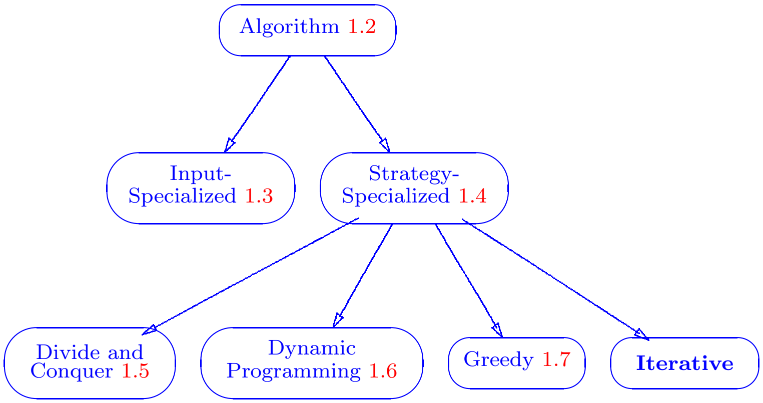
Optimal substructure: A problem is said to have optimal substructure if the optimal solution to the problem contains within it optimal solutions to the contained subproblems.

Having seen this, a greedy algorithm is simply an algorithm which makes a sequence of locally optimal decisions. Generally, the structure of the algorithm follows this pattern:

1. Repeat until the problem has been reduced to an empty or trivial base case.
   (a) Augment the solution in some locally optimal fashion.
   (b) Apply the local choice made to reduce or contract the problem.

This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy (§1.4), which gives a specific strategy to the steps of the algorithm.

1.8. Iterative Algorithm



Refinement of: Algorithm Specialized by Strategy (§1.4)

An *iterative algorithm* is an algorithm which, throughout the course of execution, maintains some approximate output. As the name implies, the primary step in the strategy is to recalculate a new approximate output based on the previous approximation. In general, the approximate output grows closer to the final output (or solution) with each iteration, but this condition is not necessary.

Another important note is that an iterative algorithm must include some termination criteria. There are many useful iterative procedures which are not algorithms without a change in their formulation. Without termination, they must be considered iterative computational methods (§1.1).

This concept if one of many known ways of narrowing the algorithm concept in terms of a strategy (§1.4), which gives a specific strategy to the steps of the algorithm.

References

[1] Donald E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Third Edition, Addison-Wesley, Reading, MA, 1997.