Computational complexity

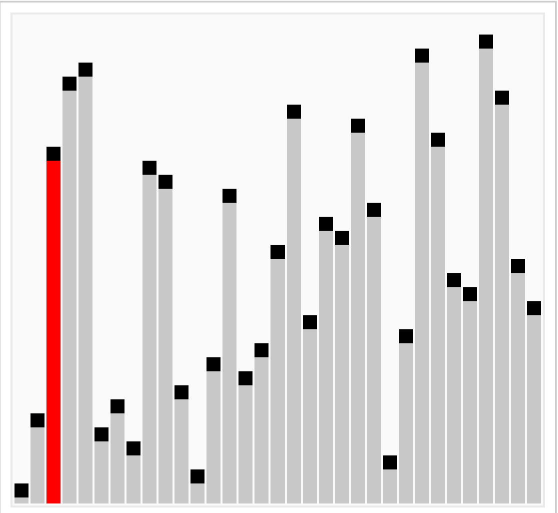
From optimization

Authors: David O'Brien, David Chen, Mark Caswell.

Contents

Introduction

Computational complexity refers to the amount of resources required to solve a type of problem by systematic application of an algorithm. Resources that can be considered include the amount of communications, gates in a circuit, or the number of processors. Because the size of the particular input to a problem will affect the amount of resources necessary, measures of complexity will have to take into account this difference. As such, they can be reported as a function of the input size. As a basic example, consider the problem of sorting an array of random numbers. Take two arrays of different sizes. Intuitively, the larger one will require more steps when using the same method. Additionally, comparisons of complexity across a range of input sizes may not be consistent – that is, an algorithm may be considered to be less complex than others for small inputs, but the same comparison may not hold for larger input sizes. Depending on starting points, iterative application of algorithms may not cover the same trajectory to the same solution even if the input to the problem is identical. One way of addressing this variability is to compare the upper and lower bounds of complexity (worst and best case scenarios, respectively) and average values. Returning to our example, consider the case when the arrays are randomly populated. There is a probability, however small, that either or both of the arrays will be populated already sorted. In this case, it is possible that the larger array needs less sorting than the smaller one.


Bubblesort animation. One of many sorting algorithms

Sorting algorithms are a good introduction to the idea of computational complexity. The problem is intuitive, and there are many different algorithms of varying complexity that can elucidate the comparisons being made, and there are many useful illustrations of the different mechanisms of sorting on the world wide web.

Machine Models

The simplest model is a deterministic Turing machine, which consists of two units: control and memory. The control unit has a finite number of states whereas the memory units has an infinite number of states in both directions of infinity. Such Turing machine M can be defined by following characteristics[4]:

1. finite set $Q \forall$ states
2. initial state $q_0 \in Q$
3. subset $F \subseteq Q$ of accepting states
4. finite set $\sum inputs$
5. finite set $A \in \sum$ tape units
6. Partial partition function

In our example of sorting number arrays, we did not consider the resources needed to completely traverse the arrays. Again, consider our example, using the following sorting mechanism: At each array location, compare the current value with the value in the next location. If they are out of order, swap them. Do this at every location until the penultimate location, and start again from the beginning. Repeat until a pass through the array is completed in which no swaps are made, meaning everything is finally sorted. There are more efficient algorithms such as mergesort, heapsort, and quicksort. The comparison between our simple algorithm and the improved ones is analogous to comparing a Turing machine to other computational models. The standard Turing machine has to take time to access all the locations on its way between two locations. This is in contrast to a random access Turing machine, in which the intermediate elements can be skipped.

Quantifying Complexity

When discussing complexity, “time” is often used as the basis for comparison. However, the “time” used is not absolute time, but rather an indication of the number of steps required to solve a problem. Because hardware is constantly improving, the absolute time required to solve a given problem will decrease, even if no change is made to the algorithm. The complexity of a problem will be identical, as it is the number of steps required, but the processing is faster on modern machines because of their processing power. Complexity should thus be quantified in a manner that is independent of hardware. It will, however, necessarily depend on the methods used, namely algorithm and machine model. A meaningful way of quantifying complexity is to state how the time scales with the size of the input.

P

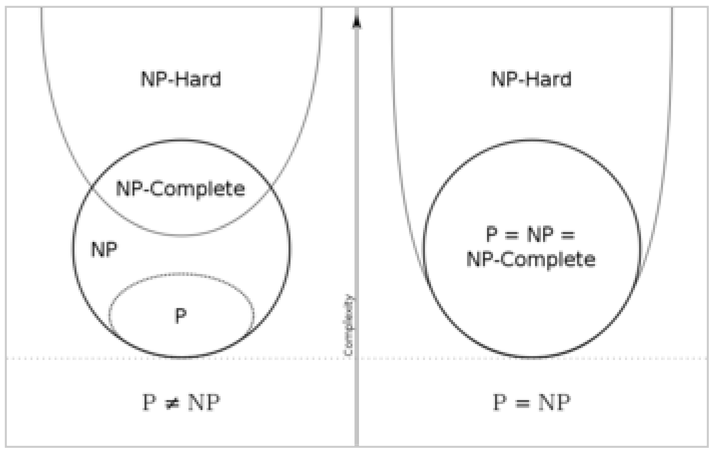
The set of problems P contains problems with whose solution-time upper bound scales as a polynomial function of the input size. Other time classes include linear time, quadratic time, or exponential time.

Cobham's thesis states that problems in P are "efficiently solvable" or "tractable." There are exceptions, but in general this is usually true.

Examples of polynomial time algorithms: the "quicksort" algorithm and basic arithmetic operations such as addition, subtraction, multiplication, division, and comparison.

NP

The set of problems NP contains problems whose solutions can be verified within a polynomially scaled upper bound. That does not mean they can be absolutely solved in polynomial time, but given a potential solution, its verity can be confirmed or denied in polynomial time.



Classification of P and NP sets

NP-Hard

A problem is NP-hard (nondeterministic polynomial time-hard) if it it can be obtained from a NP-complete problem that is polynomial time Turing-reducible. It can be said to be "at least as hard as the hardest problems in NP."

Examples of NP-hard problems: Subset sum problem, traveling salesman problem, halting problem (this last one is NP-hard but not NP-complete because all NP problems are decidable in a finite number of operations, whereas this one is not).

NP-Complete

A problem is NP-complete (nondeterministic polynomial time-complete) if it belongs to both NP as well as NP-hard. NP-complete problems can be obtained by transforming every other problem in NP in polynomial time. NP-complete problems are of note because there is an apparent correlation between the quick verifications of solutions and quick solving of the problems

Common approaches to solving NP-complete problems are heuristic algorithms and approximation algorithms.

Examples of NP-Complete problems: graph isomorphism problem, Boolean satisfiability problem, knapsack problem, Hamiltonian path problem, travelling salesman problem, subgraph isomorphism problem, and more.

P vs NP

A notable comparison is the one between the P and NP classes of problems. P and NP stand for Polynomial Time and Nondeterministic Polynomial Time, respectively. P contains decision problems that can be solved in polynomial time on a deterministic Turing machine. NP, on the other hands, contains all decision problems that, given a potential solution, can be verified in polynomial time on a deterministic Turing machine. Solution of NP problems frequently occurs in exponential time. Polynomial time means that the steps required to solve the problem has an upper bound given by a polynomial function of the input size. The P vs. NP problem asks whether every problem with solutions that can be verified in polynomial time can also be solved in polynomial time. It is one of the Millennium Prize Problems. The significance of this is that if P and NP are indeed equal, then a polynomial time solution exists for problems whose solutions can be verified in polynomial time rather than the frequently encountered exponential time solutions.

Example

As a concrete example, let us take the subset sum problem. The subset sum problem asks whether a subset of a given set can sum to zero. Take the subset [-3,-1,2,4,8]. Can a subset sum to zero? The answer is yes: [-3,-1,4]. The problem is NP-complete.

The complexity of the subset problem can be viewed as depending on two parameters, N, the number of decision variables, and P, the precision of the problem - the number of binary place values that it takes to state the problem.

If N (the number of variables) is small, then an exhaustive search for the solution is practical. If P (the number of place values) is a small fixed number, then there are dynamic programming algorithms that can solve it exactly.

Conclusions

Computational complexity is very important in analysis of algorithms. As problems become more complex and increase in size, it is important to be able to select algorithms for efficiency and solvability. The ability to classify algorithms based on their complexity is very useful. Finally, if you can solve the P vs NP problem, you will win a lot of money.

Sources

1. van Leewuen, Jan. *Handbook of Theoretical Computer Science: Algorithms and complexity, Volume 1.* Elsevier, 1990.

2. Cormen, Thomas H. *Introduction to Algorithms.* MIT Press, 2001.

3. "P vs NP Problem." Clay Mathematics Institute. Accessed May 5, 2014. http://www.claymath.org/millenium-problems/p-vs-np-problem. Last updated May 23, 2014.

4. Ding-Zhu Du. "Theory of Computational Complexity." Wiley-Interscience, 2010.