COMP 200: Elements of Computer Science
Fall 2004
Lecture 23: October 24, 2004

Upper & Lower Bounds, NP vs. P

On the Board

- Homework 5: due Monday
- Next exam: handed out next week

Algorithmic Complexity

Last lecture, we returned again to the idea that we can place lower and upper bounds on the solution of algorithmic problems. To establish an upper bound, we can exhibit an algorithm. Given some problem, an algorithm to solve the problem constructively establishes an upper bound — the problem can be solved in the given time bound. Lower bounds are established by a proof — such proofs are usually subtle and hard to construct.

Recall that an algorithm has polynomial complexity if

$$\textit{running time} < c \cdot f(\textit{size of problem})$$

where $f(N)$ is a polynomial in N and c is a constant. Algorithms that are faster than polynomial take *constant time* — that is,

$$\textit{running time} < c$$

for some constant c, independent of the size of the input set. Finding the first element of a list takes constant, or **O**(1), time. Algorithms whose running time cannot be bounded by a polynomial function are said to take *superpolynomial time*. Examples of superpolynomial functions include 2^N, $N!$, and functions that grow even faster, such as Ackerman's function.

Problems with Exponential Complexity

Some problems have complexities that are not polynomial. Consider the monkey-puzzle problems. You know about these "kids" puzzles. They consist of a picture subdivided into a square number of cards. The picture is such that each edge between two cards has a picture crossing the boundary. The goal is to discover an orientation for the cards that satisfies each boundary constraint — it lines up all the pictures correctly.

The obvious way to solve this problem is to enumerate all of the orientations. If the puzzle has *N* cards, there are *N!* orientations. Starting at one position, choose a card — there are 25 choices. The next position has 24 choices, followed by 23, and so on. Thus, total number of choices is

$$25 \cdot 24 \cdot 23 \cdot 22 \cdot 21 \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

or, for *N* cards, *N!* possibilities. The obvious algorithm takes **O**(*N!*) time.

The monkey puzzle problem is genuinely hard. *N!* grows so rapidly that we cannot solve it effectively for most values of *N*.

N	*N!*
4	24
9	362880
16	20,922,789,888,000
25	1.551121004333098e+25

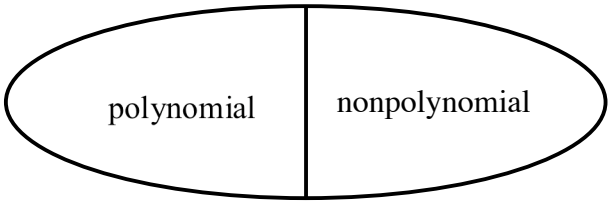
While computers keep getting faster (twice as fast every 18 to 24 months, over the last twenty-five years), that kind of growth cannot begin to keep up with **O**(*N!*).

Is there a better algorithm for the monkey puzzle problem?

> We can improve the algorithm by having it trim the choices that are infeasible — when the first *k* choices result in some misalignment of the cards.

This improved algorithm still takes **O**(*N!*) time — recall, **O** is a limit, not a tight function describing the actual behavior. In fact, any algorithm to solve this problem requires **O**(*N!*) time in the number of cards. It is a classic example of a problem that cannot be solved effectively — an ***intractable*** problem. Problems with complexity bounded by a polynomial function are considered ***tractable.***

The notion that algorithms with polynomial complexity are tractable and those with superpolynomial (greater than polynomial) complexity are intractable might create a divided universe of algorithms. Pictorially, we could represent it as:



Here, some portion of the algorithms are tractable and the rest are not.

Problems with Exponential Algorithmic Gaps

Some problems have widely separated lower and upper bounds. Problems with a complexity gap that spans the divide in our diagram — a superpolynomial upper bound and a polynomial lower bound — do not fit our model. They are not provably intractable (superpolynomial lower bound), nor are they provably tractable (known polynomial time algorithm to solve instances of the problem).

Is this issue just an academic concern? Or are there interesting problems with this property? It turns out that many problems have polynomial lower bounds and superpolynomial upper bounds. A particularly important class of problems with this property is called the NP-Complete problems. These problems have linear time lower bounds — $\mathbf{O}(N)$ — and exponential upper bounds — $\mathbf{O}(2^N)$. The Shipbuilding Problem with multiple boat lengths, mentioned a couple of lectures back, is one such problem.

Many important and practical problems fall in this class.

- Graph coloring — give an example
- Scheduling — many problems in scheduling are NP-Complete
- Hamiltonian paths — given a graph, does a path exist that visits each node in the graph exactly once. (Exponential algorithm checks all the paths.) This problem is subtle; for example, constructing a path that includes all the edges in a graph — an Eulerian path. (This problem is also called the Konigsburg Bridge problem, for the problem instance that puzzled Leonhard Euler enough to get him thinking about the general issue.)

  An Eulerian path exists for any graph that is connected and has nodes that are all of even degree (or has exactly two nodes of odd degree. These properties can be checked in polynomial time; so we can answer the yes-no question by checking these two conditions.
- Boolean satisfiability — given a logical formula over three Boolean (true or false) variables, does there exist an assignment of values to the variables that makes the formula true? [The formula consists of individual terms, connected by the logical operators *and, or, not,* and

implies.] The best known algorithm requires 2^N time, where N is the number of terms (elementary assertions) in the formula. [This metric is roughly equivalent to counting the operators.]

What makes these problems interesting? First, many real-world problems have these issues buried inside. Classroom scheduling at a university — assigning each class meeting to a room and an hour — can be modeled as a scheduling problem or as a graph coloring problem. Constructing a building, finishing a COMP 200 final project, and laying out the timetable for a college ping-pong tournament are all scheduling problems. Path problems, such as the Hamiltonian path problem, play an important role in scheduling package delivery. Coloring and path problems arise in laying out electronic components on an integrated circuit (and figuring out where to run the microscopic wires — or *traces* — that connect them.

Intellectually, these problems provide one of the biggest open puzzles in Computer Science. Because the lower and upper bounds span the boundary between tractable and intractable, we do not know if we should give up and approximate, or keep looking for polynomial-time algorithms. In fact, the most interesting property (to my mind) of the set of NP-Complete problems is that they are all intimately related. Every problem in NPC (the class of NP-Complete problems) is polynomially related to the rest of them. That is, there exists a polynomial time transformation on the input of the problem that makes it an appropriate input for some other problem in NPC, *and*, an inverse translation exists to take the input for that problem to our original problem. We call this property *reducibility* of problems in NPC.

To prove that your problem is in NPC, you must show such two reductions — your problem to some other problem already known to be in NPC and from that problem back to your problem. The reductions must take polynomial time, so that a polynomial solution to the other problem would work as a polynomial solution to your problem — translate in, solve, translate out. This bidirectional reducibility justifies the word "Complete" in NP-Complete.

We also know of problems that are as hard as those in NP, but are not part of the Complete set. To show that a problem is NP-hard, we just need to show that it can solve some NPC problem. We can exhibit a one-way reduction to show this property.