Good Algorithms

Recall from lecture 1

I Design of Algorithms

II Analysis of Algorithms

III Lower Bounds – do we have the best algorithm?

Lower bounds. Problem P, Algorithm A. Run-time $t(n)$.

When is Algorithm A good enough?

e.g. branch and bound $O(2^n)$ or $O(n!)$

Want to show that any algorithm for problem P has worst case run-time $\geq t(n)$ asymtotically

i.e. $\Omega(t(n))$

Such lower bounds are hard to prove.

Lower bound techniques

- based on output size. e.g. computing all permutations of $1 \cdot\cdot n$ takes $\Omega(n!)$
- information-theoretic lower bounds

  e.g. search for item a in sorted list $a_1 \cdot\cdot\cdot a_n$.

  Must distinguish n possibilities and each comparison gives 1 bit of information. Thus $\Omega(\log n)$ worst case.
- adversary arguments

  for any algorithm, the adversary makes a hard input – like a game
- reductions – we'll use these

e.g. on assign 2 you reduced multiplying $n \times n$ matrices to squaring $n \times n$ matrices

Thus run-time for squaring is a lower bound on run-time for multiplying.

[e.g. on assign 3 you reduced finding shortest path with fewest edges to shortest path.]

State of the Art in Lower Bounds/Impossibility Results

- Some problems don't have algorithms
  Turing 1930's. We'll cover this at end of course.
  Also in CS 245 and CS 360.
- Some problems can only be solved in exponential time.
- Some problems have like $\Omega(n \log n)$ lower bounds on restricted model of computing, e.g. sorting.

Major Open Question

There are many problems e.g. Travelling Salesman, 0-1 Knapsack, where no one knows a poly. time alg. and no one can prove there's no poly. time alg.

The best we can do: prove that a large set of problems are <u>equivalent</u> in the sense that a poly. time alg. for one yields a poly time alg. for all.

That's a general overview. Now fill this in.

Polynomial Time = Efficient

polynomial time means [worst case] running time is $O(n^k)$ for some constant k. n = input size

e.g. $\Theta(2^n)$, $\Theta(n!)$ are NOT poly. time.

Most of the algs. we've studied have been poly. time, except backtracking, branch-and-bound, pseudo-poly. time alg. for 0-1 knapsack $O(n \cdot W)$.

polynomial time = "good"

– quote from Edmonds [from Schrijver's book] in his 1963 paper "Paths, Trees and Flowers".

2. Digression. An explanation is due on the use of the words "efficient algorithm." First, what I present is a conceptual description of an algorithm and not a particular formalized algorithm or "code."

For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, "efficient" means "adequate in operation or performance." This is roughly the meaning I want—in the sense that it is conceivable for maximum matching to have no efficient algorithm. Perhaps a better word is "good."

I am claiming, as a mathematical result, the existence of a *good* algorithm for finding a maximum cardinality matching in a graph.

There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether *or not* there exists an algorithm whose difficulty increases only algebraically with the size of the graph.

The mathematical significance of this paper rests largely on the assumption that the two preceding sentences have mathematical meaning. I am not prepared to set up the machinery necessary to give them formal meaning, nor is the present context appropriate for doing this, but I should like to explain the idea a little further informally. It may be that since one is customarily concerned with existence, convergence, finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

History:

- success of linear programming - simplex method
  Dantzig 1963
- integer programming (algs. not as good!)
- reducing problems to Integer programming as
  a way to solve them.

(In fact integer programming is NP-hard,
but linear programming has an efficient alg.)

Reduction

Problem A reduces [in poly. time] to problem B if a [poly. time] algorithm for B can be used to get a [poly. time] algorithm for A.

i.e. there is a [poly. time] algorithm for A that makes subroutine calls to a [poly. time] algorithm for B.

Note: we don't need to have such an algorithm for B.

notation $A \le B$ for poly. time $A \le_p B$

"A is easier than B"

Consequences of $A \le_p B$:

A lower bound for A [A cannot be solved in poly. time] yields a lower bound for B [B cannot " "]

Even if we don't have an alg. for B OR a lower bound for A, we can still use reductions to show that problems are equivalently hard (show $A \le_p B$, $B \le_p A$)

Example

A = multiplying integers

We saw an $O(n^{1.59})$ alg. (counting bits)

Best lower bound $\Omega(n)$

B = squaring integers

Obviously, squaring is easier than multiplying because it's a special case.

squaring $\le_P$ multiplying

thus

- a faster multiplication alg. gives a faster squaring alg.
- a lower bound for squaring gives a lower bd. for multiplying

---

multiplying $\le_P$ squaring

$$x \cdot y = \frac{1}{4}\left((x+y)^2 - (x-y)^2\right)$$

this reduction takes $O(n)$ time $n = \#$ bits in x, y

- compute $x+y$, $x-y$
- hand to squaring routine
- subtract, divide by 4.

thus

- a faster squaring alg. gives a faster multiplying alg.
- a lower bound for multiplying gives a lower bd. for squaring

Decision Problems

- problems where output is YES/NO

Theory of NP-completeness focusses on decision problems
- it's easier that way
- optimization and decision are usually equivalent wrt poly. time.

Examples
- given a number, is it prime?
- given a graph, does it have a Hamiltonian cycle?
- given an edge weighted graph and number k, does it have a TSP tour of length $\le k$?

Equivalence of optimization and decision
- no general proof, but usually OK

e.g. max ind. set
- optimization - find max ind. set
- decision - given k, is max ind. set $\ge k$.

decision $\le_p$ optimization
- just see if the max ind. set is $\ge k$

optimization $\le_p$ decision
- find max size by asking for each $k = 1 .. n$
- find actual max ind. set by throwing away vertices one by one and repeatedly asking decision.

Equivalence is not always known

e.g. testing if a number is prime or composite seems easier than finding its prime factorization

e.g. (going the other way)
although we can find MST for points in plane with Euclidean distances in poly. time, it's not known how to test if weight(MST) $\le k$
(measuring bit complexity

$$\sum \sqrt{n_i} \leq k \quad ? \qquad n_i \in \mathbb{N}$$

OPEN – do this in poly. time)