Complexity

Some problems seem to be more difficult to solve efficiently than others.

- the obvious brute force algorithm often has very different running time for different algorithms
  - e.g. closest pair of points – n^2
    - compute the distance for every pair
  - e.g. 0-1 knapsack – 2^n
    - try every subset

Complexity

Some problems seem to be more difficult to solve efficiently than others.

- small changes in a problem can make it much harder to solve
  - e.g. fractional knapsack vs 0-1 knapsack
  - e.g. linear programming vs integer linear programming
  - e.g. shortest path in a graph vs the longest
    - (note: general graph, not limited to DAG)
  - e.g. use every edge once (Euler circuit) vs use every vertex once (hamiltonian cycle)

Complexity

Some problems seem to be more difficult to solve efficiently than others.

- algorithmic techniques which work to speed up some problems don't work for others
  - e.g. greedy vs dynamic programming vs recursive backtracking

Complexity

Are there some problems which take fundamentally longer to solve than others, or have we just not been clever enough yet to find an efficient solution?

Famous Complexity Classes

P – decision problems solvable by a deterministic Turing machine in polynomial time

NP – decision problems verifiable by a deterministic Turing machine in polynomial time

FP – function problems solvable by a deterministic Turing machine in polynomial time

FNP – function problems verifiable by a deterministic Turing machine in polynomial time

Decision Problems vs Function Problems

Decision problems are problems where the result is a yes/no answer.

- e.g. is there a solution to the 0-1 knapsack problem with total weight ≤ W and total value ≥ V?

Function problems compute the result of a function.

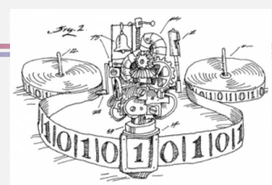
- e.g. 0-1 knapsack problem: maximize the total value such that the total weight ≤ W

Observation –

- a function problem can be solved efficiently given a black box for the corresponding decision problem
  - "efficiently" = logarithmic number of steps
    - use one-sided binary search

(one-sided binary search for knapsack means trying $V = 2^i$ for i = 0, 1, 2, … until the answers are different for successive values of i, then repeating the process within the interval found to find a smaller interval, and so forth)

Turing Machines



http://www.nikoloplakis.gr/

A *Turing machine* is a theoretical machine consisting of:

- an infinite tape divided into cells
- a head that can read and write symbols on the tape, and move one cell left or right
- a current state, which is one of a finite number of possible states
- a finite table which, given a current state and symbol on the tape, specifies an action (erase or write symbol), a movement (left, right, or stay), and a new state

Tape symbol	Current state A			Current state B			Current state C		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state
0	1	R	**B**	1	L	**A**	1	L	**B**
1	1	L	**C**	1	R	**B**	1	R	**HALT**

Deterministic vs Nondeterministic

A *deterministic* Turing machine has at most one rule that applies to a given state and symbol.

A *nondeterministic* Turing machine may have multiple rules that apply to a given state and symbol.

Famous Complexity Classes

P – solvable by a deterministic Turing machine in polynomial time

NP – verifiable by a deterministic Turing machine in polynomial time

- alternatively, solvable by a nondeterministic Turing machine in polynomial time

Key points –

- for NP, technically it is only "yes" solutions that are polynomial-time verifiable
  a "yes" answer requires only a single instance that works (and is checkable in polynomial time)
  a "no" answer requires showing that no instance works
- in both cases, there are at most a polynomial number of choices to make in order to generate the solution
  - for each choice –
    - deterministic has rules to pick the right alternative
    - nondeterministic can be thought of as correctly guessing the right alternative

Famous Complexity Classes

- does NP include P? that is, is every problem in P also in NP?
  - yes – if you can solve a problem in polynomial time, you can always verify a possible solution by computing the solution yourself and comparing
- are there problems in NP that aren't in P?
  - probably
  - (proving this one way or the other will get you fame and a million dollars)
- are there problems that aren't in NP?
  - yes e.g. function problems (NP is only decision problems), the halting problem (undecidable)

Famous Complexity Classes

P – decision problems solvable by a deterministic Turing machine in polynomial time

NP – decision problems verifiable by a deterministic Turing machine in polynomial time

FP – function problems solvable by a deterministic Turing machine in polynomial time

FNP – function problems verifiable by a deterministic Turing machine in polynomial time

Famous Complexity Classes

- does FNP contain FP?
  - yes
- are there problems in FNP that aren't in FP?
  - probably (for the same reason as there are probably problems in NP not in P)
- are there problems that aren't in FNP?
  - yes – e.g. enumeration tasks (solution size can be exponential)

Famous Complexity Classes

- is FP easier or harder than P?
  - no – each can be used a black box to efficiently solve the other problem
    A can't be easier than B if A can be used to efficiently solve B
    - the solution to the FP version can be used directly to answer the P version's question
    - the P version can be used as a black box to find the FP solution in polynomial time using one-sided binary search
- is FNP easier or harder than NP?
  - [Bellare, Goldwasser 1994] under certain assumptions, there are FNP problems that are harder than their corresponding NP problems
    i.e. there seem to be problems in FNP where a solution to the NP version can't be used to efficiently solve the FNP version

Determining Complexity

Reductions are useful for making arguments about complexity.

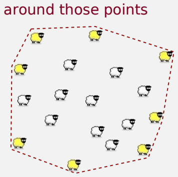
Let A be a problem with a polynomial-time reduction to B.
- i.e. polynomial time to turn an instance of A into an instance of B, and polynomial time to turn a solution for B into a solution for A

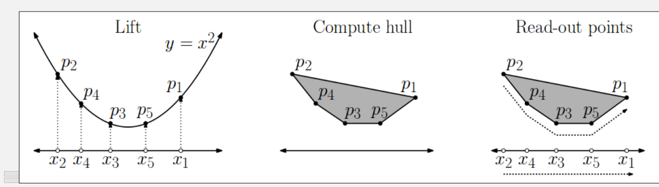
Then B is at least as hard as A. easy/hard has to do with efficiency of solution
Why?
- if B has an efficient algorithm, A can be solved efficiently via the reduction
- if B doesn't have an efficient algorithm, it may still be possible to solve A efficiently using a different approach – we don't know

Reductions for Lower Bounds

Sorting can be reduced to convex hull –
- for each element *i* to be sorted, create a point (i,i^2)
- compute the convex hull of the points
  - (using an algorithm that outputs the hull points in cyclic order)
- read points on the hull from left to right, starting with the leftmost point in the hull
  - this is the sorted order of the elements

the convex hull of a set of points is the shape of a rubber band stretched around those points





http://www.cs.umd.edu/class/fall2014/cmsc754/Lects/lect04.pdf

Reductions for Lower Bounds

Sorting can be reduced to convex hull –
- for each element *i*, create a point (i,i^2) O(n)
- compute the convex hull O(??)
  - (using an algorithm that outputs the hull points in cyclic order)
- read points on the hull from left to right, starting with the leftmost point in the hull O(n)

Since comparison-based sorting is known to take Ω(n log n) time, the ?? step cannot be faster than n log n or else we'd have a better algorithm for sorting using convex hull.

→ convex hull (if the points on the hull are output in cyclic order) is Ω(n log n)

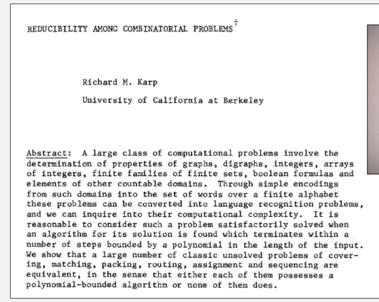
Completeness

Within a class, the *complete* problems are the hardest – if you can solve a complete problem, you can solve every problem in the class.

- **P-complete** – set of problems in P such that every other problem in P is polynomial-time reducible to one in the set
  - these are problems believed to be “inherently sequential” i.e. a parallel computer would not significantly speed them up
- **NP-complete** – set of problems in NP such that every other problem in NP is polynomial-time reducible to one in the set

Karp's 21 NP-Complete Problems

One of the first demonstrations that many common computational problems are computationally intractable. (1972)

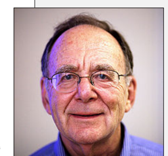


REDUCIBILITY AMONG COMBINATORIAL PROBLEMS†

Richard M. Karp

University of California at Berkeley

Abstract: A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.



Richard Karp, 1935-

American computer scientist

known for work in computer science, combinatorial algorithms, operations research, bioinformatics

- Held-Karp algorithm – TSP
- Edmonds-Karp algorithm – max flow
- 21 NP-complete problems
- Hopcroft-Karp algorithm – matchings in bipartite graphs
- Karp-Lipton theorem – complexity result
- Rabin-Karp string search algorithm

1985 Turing Award for contributions to the theory of NP-completeness

Karp's 21 NP-Complete Problems

clique	is there a set of k vertices in the graph such that every vertex in the set is connected to every other vertex in the set?
clique cover	can the graph be partitioned into k cliques?
vertex cover	is there a set of k vertices in the graph such that every edge has at least one endpoint in the set?
chromatic number	can the graph be colored with k colors?
feedback node set	is there a set of k vertices in an undirected graph whose removal leaves the graph without cycles?
feedback arc set	is there a set of k edges in a directed graph whose removal leaves the graph without directed cycles?
directed hamiltonian cycle	is there a directed/undirected cycle which visits every vertex exactly once?
undirected hamiltonian cycle	
max cut	can the vertices of a graph be split into two sets so that the sum of the weights of the edges between vertices in different sets is at most k?
Steiner tree	version of MST where additional points may be introduced to reduce the overall weight of the tree

Karp's 21 NP-Complete Problems

CNFSAT	is there an assignment of values to make a boolean expression with only OR and NOT within a clause and clauses joined by AND true?
3-SAT	CNFSAT where there are exactly three variables per clause
binary integer programming	linear programming where variables are constrained to the values 0 or 1
set packing	in a collection of sets, is there a group of k that are disjoint?
set covering	given a collection of subsets of X, is there a group of k subsets that together contain every element of X?
exact cover	given a collection of subsets of X, is there a group of those subsets such that every element of X is contained in exactly one subset?
hitting set	given a collection of subsets of X, is there a subset H of X of size k so that every set in the collection contains at least one element of H?
3-dimensional matching	given a set of triples (x,y,z) where $x \in X$, $y \in Y$, $z \in Z$, is there a collection of triples such that every element of X, Y, and Z occurs exactly once?
0-1 knapsack	is there a set of items with total weight $\leq W$ and total value $\geq V$?
partition	can a set of numbers be split into two parts so that the sums of the parts are equal?
job sequencing	can a set of jobs be scheduled so that no more than k miss their deadlines?

Proving NP-Completeness

Most NP-complete problems are proven NP-complete by a reduction to a known NP-complete problem.

If problem A has a polynomial-time reduction to problem B, which of the following can you conclude?

- **if A is NP-complete, there can't be a polynomial-time algorithm for B**
- if B is NP-complete, there can't be a polynomial-time algorithm for A
- none of the above