CS 440 Theory of Algorithms / CS 468 Algorithms in Bioinformatics

Limitations of Algorithm Power

Lower Bounds

Lower bound: **an estimate on a minimum amount of work needed to solve a given problem**

Examples:

- **Number of comparisons needed to find the largest element in a set of *n* numbers**
- **Number of comparisons needed to sort an array of size *n***
- **Number of comparisons necessary for searching in a sorted array**
- **Number of multiplications needed to multiply two *n*-by-*n* matrices**

Lower Bounds (cont.)

- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- ***Tight*** lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower bound	Tightness
Sorting	$\Omega(n\log n)$	yes
Searching in a sorted array	$\Omega(\log n)$	yes
Element uniqueness	$\Omega(n\log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
Multiplication of n-by-n matrices	$\Omega(n^2)$	unknown

Methods for Establishing Lower Bounds

- Trivial lower bounds

- Information-theoretic arguments (decision trees)

- Adversary arguments

- Problem reduction

Trivial Lower Bounds

Trivial lower bounds: **based on counting the number of items that must be processed in input and generated as output**

Examples

- **Finding max element**
- **Polynomial evaluation**
- **Sorting**
- **Element uniqueness**
- **Hamiltonian circuit existence**

Conclusions

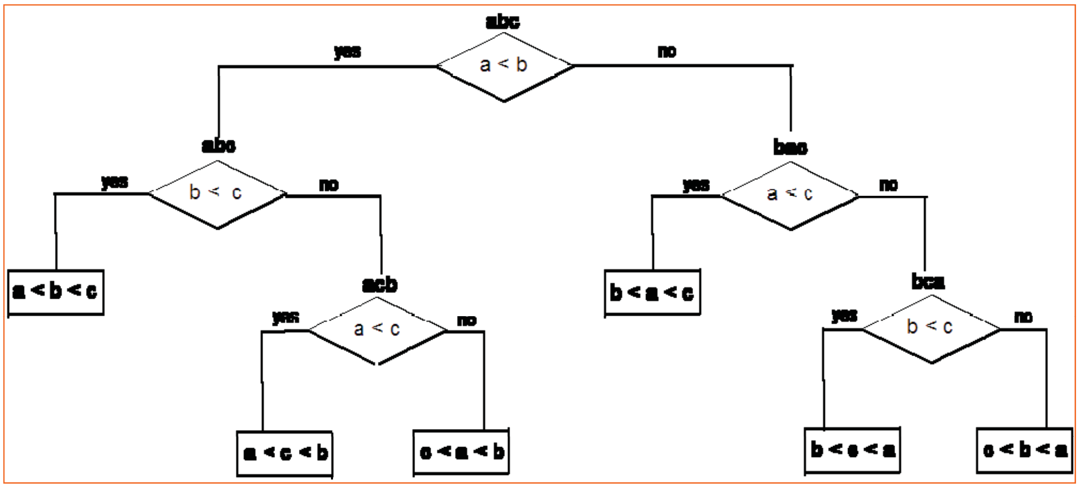
- **May and may not be useful**
- **Be careful in deciding how many elements <u>must</u> be processed**

Decision Trees

Decision tree **— a convenient model of algorithms involving comparisons in which:**

- **Internal nodes represent comparisons**
- **Leaves represent outcomes**

Decision tree for 3-element insertion sort



Decision Trees and Sorting Algorithms

- **Any comparison-based sorting algorithm can be represented by a decision tree**
- **Number of leaves (outcomes) = $n!$**
- **Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$**
- **Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm**
- $\lceil \log_2 n! \rceil \approx n \log_2 n$
- **This lower bound is tight**

Adversary Arguments

***Adversary argument*: a method of proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input. The adversary cannot lie, however.**

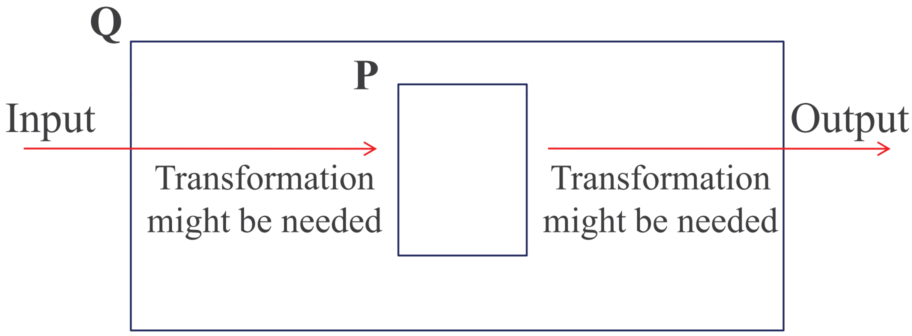
Example 1: "Guessing" a number between 1 and n with yes/no questions (e.g., are 4 questions enough to guess a number between 1 and 17)
Adversary: Puts the number in a larger of the two subsets generated by last question

Example 2: Merging two sorted lists of size n
$a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$
Adversary: $a_i < b_j$ iff $i < j$
Output $b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$ requires $2n$-1 comparisons of adjacent elements

Lower Bounds by Problem Reduction

Idea: If problem *P* is at least as hard as problem *Q*, then a lower bound for *Q* is also a lower bound for *P*.

Hence, find problem *Q* with a known lower bound that can be reduced to problem *P* in question.



Does the other way around, i.e., using Q with a known lower bound to solve P, show the lower bound of P?

Lower Bounds by Problem Reduction

Example:

***P*: finding convex hull for *n* points in Cartesian plane**

***Q*: comparison-based sorting problem (known to be in $\Omega(n\log n)$)**

Show that convex hull problem is in $\Omega(n\log n)$

Example: Is squaring large integers simpler than multiplying large integers?

- $x^2 = x \times x$

  We can use multiply to do square → what does it tell us about the complexity of the two operations?
- $x \times y = ((x + y)^2 - (x - y)^2) / 4$

  We can use square to do multiply → what do we know now?

➔ They have the same complexity

Our old list of problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Classifying Problem Complexity

Is the problem *tractable*, i.e., is there a polynomial-time (O(*p*(*n*)) algorithm that solves it?

Possible answers:

- **Yes**
- **No**
  - **because it's been proved that no algorithm exists at all (e.g., Turing's *halting problem*)**
  - **because it's been be proved that any algorithm takes exponential time → intractable**
- **Unknown**
- **Unknown, but if such algorithm were to be found, then it would provide a means of solving many other problems in polynomial time**

Problem Types: Optimization and Decision

- ***Optimization problem***: find a solution that maximizes or minimizes some objective function
- ***Decision problem:*** answer yes/no to a question
  - A correct algorithm that solves a decision problem *accepts* the "yes-instances" and *rejects* the "no-instances."

Many problems have decision and optimization versions.

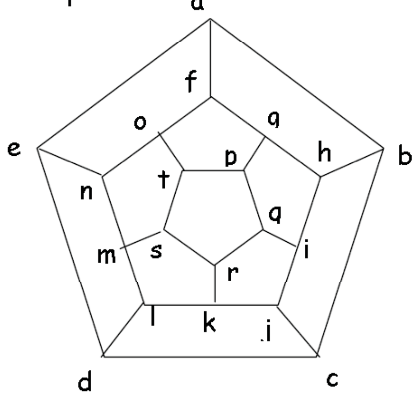
E.g.: traveling salesman problem

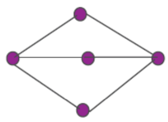
- ***optimization***: find Hamiltonian cycle of minimum length
- ***decision***: given an additional bound m find Hamiltonian cycle of length $\le m$

Decision problems are more convenient for formal investigation of their complexity.

Problem Types: Optimization and Decision

- **Example: Hamiltonian cycle problem (HAM)**
  - "Find a Hamiltonian cycle in an undirected graph G." is not a decision problem.
  - But it can be rephrased into a decision problem: "Does graph G contain a Hamiltonian cycle?" (HAMD)
  - And, given a path, it is easy to answer that "Is this path a Hamiltonian cycle?"
    - Examples





Some more problems

- ***Partition***: Given *n* positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum
- ***Bin packing***: given *n* items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1
- ***Graph coloring***: For a given graph find its chromatic number, ie, the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color
- ***CNF satisfiability***: Given a boolean expression in conjunctive normal form (conjunction of disjunctions of literals), is there a truth assignment to the variables that makes the expression true?

Class *P*

P: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem's input size n

Examples:

- Searching
- Element uniqueness
- Graph connectivity
- Graph acyclicity
- Primality testing (finally proved in 2002)

Class *NP*

NP (*nondeterministic polynomial*): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

A *nondeterministic polynomial algorithm* is an abstract two-stage procedure that:

- Nondeterministic ("guessing") stage:
  Generates a random string purported to solve the problem
- Deterministic ("verification") stage:
  Checks whether this solution is correct in polynomial time

By definition, an NP algorithm solves the problem if it's capable of generating and verifying a solution on one of its tries in polynomial time

Example: CNF satisfiability

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true

Example: (A v ¬B v ¬C) & (A v B) & (¬B v ¬D v E) & (¬D v ¬E)

v: boolean OR
&: boolean AND

Truth assignments:

A B C D E
0 0 0 0 0
. . .
1 1 1 1 1

How many?

Checking phase: O(*n*)

What problems are in *NP*?

- **CNF-SAT**
- **Hamiltonian circuit existence**
- **Partition problem: Is it possible to partition a set of *n* integers into two disjoint subsets with the same sum?**
- **Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)**

- **All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:**

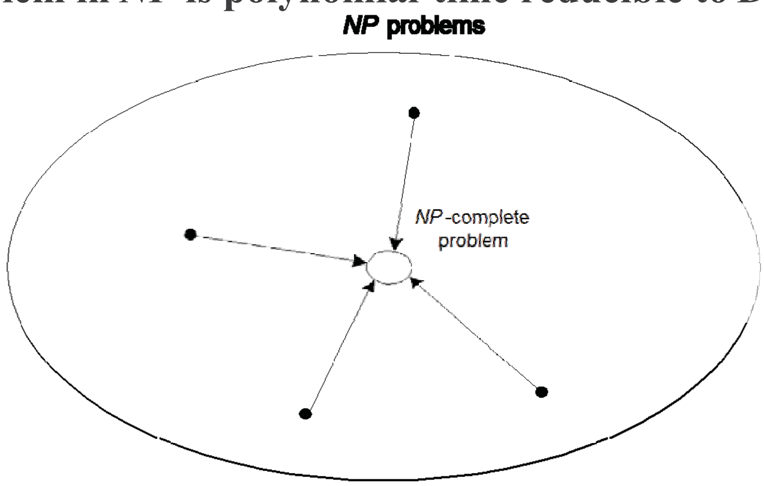
$$P \subseteq NP$$

- **Big question: $P = NP$?**

NP-Complete Problems

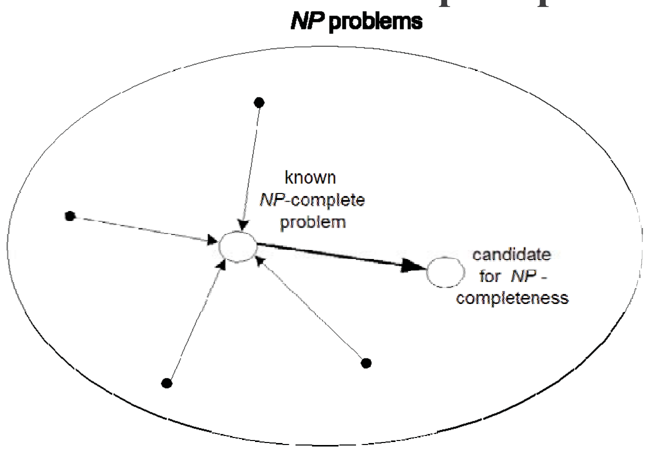
A decision problem *D* is <u>*NP*-complete</u> if it's as hard as any problem in *NP*, i.e.,

- ***D* is in *NP***
- **Every problem in *NP* is polynomial-time reducible to *D***



NP-Complete Problems (cont.)

Other *NP*-complete problems are obtained through polynomial-time reductions from a known *NP*-complete problem



But we need to have the first NP-Complete problem to start:
Cook's theorem (1971): CNF-SAT is *NP*-complete

P = *NP* ? Dilemma Revisited

- ***P* = *NP* would imply that every problem in *NP,* including all *NP*-complete problems, could be solved in polynomial time**

- **If a polynomial-time algorithm for just one *NP*-complete problem is discovered, then every problem in *NP* can be solved in polynomial time, i.e., *P* = *NP***

- **Most but not all researchers believe that *P* ≠ *NP* , i.e. *P* is a proper subset of *NP***