CS 573: Algorithms

Sariel Har-Peled
sariel@illinois.edu
3306 SC

University of Illinois, Urbana-Champaign

Fall 2014

Administrivia, Introduction

Lecture 1
August 26, 2014

The word "algorithm" comes from...

Muhammad ibn Musa al-Khwarizmi
780-850 AD
The word "algebra" is taken from the title of one of his books.

Part I

Administrivia

Instructional Staff

1. Instructor:
   - Sariel Har-Peled (`sariel`)
2. Teaching Assistants:
   2.1 Vivek Madan
3. Office hours: See course webpage
4. Email: See course webpage

Online resources

1. Webpage:
   `courses.engr.illinois.edu/cs573/fa2014/`
   General information, homeworks, etc.
2. Moodle: Quizzes, solutions to homeworks.
3. Online questions/announcements: Piazza
   Online discussions, etc.

Textbooks

1. Prerequisites: CS 173 (discrete math), CS 225 (data structures) and CS 373 (theory of computation)
2. Recommended books:
   2.1 Algorithms by Dasgupta, Papadimitriou & Vazirani. Available online for free!
   2.2 Algorithm Design by Kleinberg & Tardos
3. Lecture notes: Available on the web-page before/during/after every class.
4. Additional References
   4.1 Previous class notes of Jeff Erickson, Sariel Har-Peled and the instructor.
   4.2 Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein.
   4.3 Computers and Intractability: Garey and Johnson.

Prerequisites

1. Asymptotic notation: $O(), \Omega(), o()$.
2. Discrete Structures: sets, functions, relations, equivalence classes, partial orders, trees, graphs
3. Logic: predicate logic, boolean algebra
4. Proofs: **by induction**, by contradiction
5. Basic sums and recurrences: sum of a geometric series, unrolling of recurrences, basic calculus
6. Data Structures: arrays, multi-dimensional arrays, linked lists, trees, balanced search trees, heaps
7. Abstract Data Types: lists, stacks, queues, dictionaries, priority queues
8. Algorithms: sorting (merge, quick, insertion), pre/post/in order traversal of trees, depth/breadth first search of trees (maybe graphs)
9. Basic analysis of algorithms: loops and nested loops, deriving recurrences from a recursive program
10. Concepts from Theory of Computation: languages, automata, Turing machine, undecidability, non-determinism
11. Programming: in some general purpose language
12. Elementary Discrete Probability: event, random variable, independence
13. Mathematical maturity

Grading Policy: Overview

1. Attendance/clickers: 5%
2. Quizzes: 5%
3. Homeworks: 15%
4. Midterm: 30%
5. Finals: 45% (covers the full course content)

Homeworks

1. One quiz every 1-2-3 weeks: Due by midnight on Sunday.
2. One homework every 1-2-3 weeks.
3. Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
   3.1 Short quiz-style questions to be answered individually on *Moodle*.
4. Groups can be changed a *few* times only.

More on Homeworks

1. No extensions or late homeworks accepted.
2. To compensate, the homework with the least score will be dropped in calculating the homework average.
3. Important: Read homework FAQ/instructions on website.

Advice

1. Attend lectures, please ask plenty of questions.
2. Clickers...
3. Don't skip homework and don't copy homework solutions.
4. Study regularly and keep up with the course.
5. Ask for help promptly. Make use of office hours.

Homeworks

1. Homework 0 is posted on the class website. Quiz 0 available
2. Homework 0 to be submitted in individually.

Part II

Course Goals and Overview

Topics

1. Some fundamental algorithms
2. Broadly applicable techniques in algorithm design
   2.1 Understanding problem structure
   2.2 Brute force enumeration and backtrack search
   2.3 Reductions
   2.4 Recursion
      2.4.1 Divide and Conquer
      2.4.2 Dynamic Programming
   2.5 Greedy methods
   2.6 Network Flows and Linear/Integer Programming (optional)
3. Analysis techniques
   3.1 Correctness of algorithms via induction and other methods
   3.2 Recurrences
   3.3 Amortization and elementary potential functions
4. Polynomial-time Reductions, NP-Completeness, Heuristics

Goals

1.
2. Learn/remember some basic tricks, algorithms, problems, ideas
3. Understand/appreciate limits of computation (intractability)
4. Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
5. Have fun!!!

Part III

Algorithms and efficiency

Primality testing

Problem

Given an integer $N > 0$, is N a prime?

SimpleAlgorithm:

```
    for i = 2 to ⌊√N⌋ do
        if i divides N then
            return ``COMPOSITE''
    return ``PRIME''
```

Correctness? If N is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$

Running time? $O(\sqrt{N})$ divisions? Sub-linear in input size!

Wrong!

Primality testing

...Polynomial means... in input size

How many bits to represent N in binary? $\lceil \log N \rceil$ bits.

Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.
Exponential in the input size $n = \log N$.

1. Modern cryptography: binary numbers with 128, 256, 512 bits.
2. Simple Algorithm will take 2^{64}, 2^{128}, 2^{256} steps!
3. Fastest computer today about 3 petaFlops/sec: 3×2^{50} floating point ops/sec.

Lesson:

Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

Efficient algorithms

So, is there an *efficient/good/effective* algorithm for primality?

Question:

What does efficiency mean?

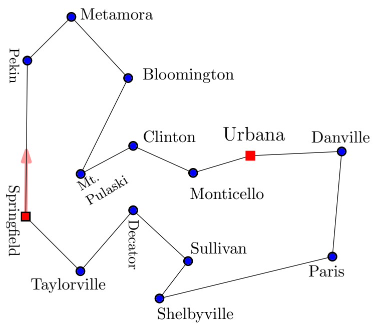
In this class *efficiency* is broadly equated to *polynomial time*. $O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where n is size of the input.

Why? Is n^{100} really efficient/practical? Etc.

Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

problem

Lincoln's tour



1. Circuit court - ride through counties staying a few days in each town.
2. Lincoln was a lawyer traveling with the Eighth Judicial Circuit.
3. Picture: travel during 1850.
   3.1 Very close to optimal tour.
   3.2 Might have been optimal at the time..

Solving by a Computer

Is it hard?

1. n = number of cities.
2. n^2: size of input.
3. Number of possible solutions is

$$n * (n-1) * (n-2) * \ldots * 2 * 1 = n!.$$

4. $n!$ grows very quickly as n grows.
   $n = 10$: $n! \approx 3628800$
   $n = 50$: $n! \approx 3 * 10^{64}$
   $n = 100$: $n! \approx 9 * 10^{157}$

Solving by a Computer

Fastest computer...

1. Fastest super computer can do (roughly)

$$2.5 * 10^{15}$$

operations a second.

2. Assume: computer checks $2.5 * 10^{15}$ solutions every second, then...
   2.1 $n = 20 \implies$ 2 hours.
   2.2 $n = 25 \implies$ 200 years.
   2.3 $n = 37 \implies 2 * 10^{20}$ years!!!

What is a good algorithm?

Running time...

Input size	n^2 ops	n^3 ops	n^4 ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
100	0 secs	0 secs	0 secs	never
8000	0 secs	0 secs	1 secs	never
16000	0 secs	0 secs	26 secs	never
32000	0 secs	0 secs	6 mins	never
64000	0 secs	0 secs	111 mins	never
200,000	0 secs	3 secs	7 days	never
2,000,000	0 secs	53 mins	202.943 years	never
10^8	4 secs	12.6839 years	10^9 years	never
10^9	6 mins	12683.9 years	10^{13} years	never

What is a good algorithm?

Running time...

Primes is in **P**!

Theorem (Agrawal-Kayal-Saxena'02)

There is a polynomial time algorithm for primality.

First polynomial time algorithm for testing primality. Running time is $O(\log^{12} N)$ further improved to about $O(\log^6 N)$ by others. In terms of input size $n = \log N$, time is $O(n^6)$.

What about before 2002?

Primality testing a key part of cryptography. What was the algorithm being used before 2002?
Miller-Rabin *randomized* algorithm:

1. runs in polynomial time: $O(\log^3 N)$ time
2. if N is prime correctly says "yes".
3. if N is composite it says "yes" with probability at most $1/2^{100}$ (can be reduced further at the expense of more running time).

Based on Fermat's little theorem and some basic number theory.

Factoring

1. Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
2. Relies on the difficulty of factoring a composite number into its prime factors.
3. There is a polynomial time algorithm that decides whether a given number N is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

Lesson

Intractability can be useful!

Unit-Cost RAM Model

Informal description:

1. Basic data type is an integer/floating point number
2. Numbers in input fit in a word
3. Arithmetic/comparison operations on words take constant time
4. Arrays allow random access (constant time to access $A[i]$)
5. Pointer based data structures via storing addresses in a word

Example

Sorting: input is an array of n numbers

1. input size is n (ignore the bits in each number),
2. comparing two numbers takes $O(1)$ time,
3. random access to array elements,
4. addition of indices takes constant time,
5. basic arithmetic operations take constant time,
6. reading/writing one word from/to memory takes constant time.

We will usually not allow (or be careful about allowing):

1. bitwise operations (and, or, xor, shift, etc).
2. floor function.
3. limit word size (usually assume unbounded word size).

Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in several important situations so one has to be careful.

1. For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two n-digit numbers, primality etc.
2. Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by 2^k where k is word length.
3. Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.

Models used in class

In this course:

1. Assume unit-cost RAM by default.
2. We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.

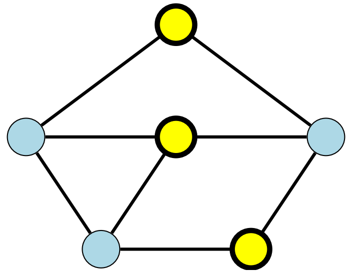
Part IV

Reductions

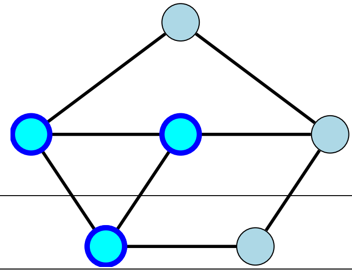
Independent Sets and Cliques

Given a graph G, a set of vertices V' is:

1. An ***independent set***: if no two vertices of V' are connected by an edge of G.



2. ***clique***: *every* pair of vertices in V' connected by an edge of G.



The **Independent Set** and **Clique** Problems

Independent Set

Instance: A graph **G** and an integer k.
Question: Does **G** has an independent set of size $\geq k$?

Clique

Instance: A graph **G** and an integer k.
Question: Does **G** has a clique of size $\geq k$?

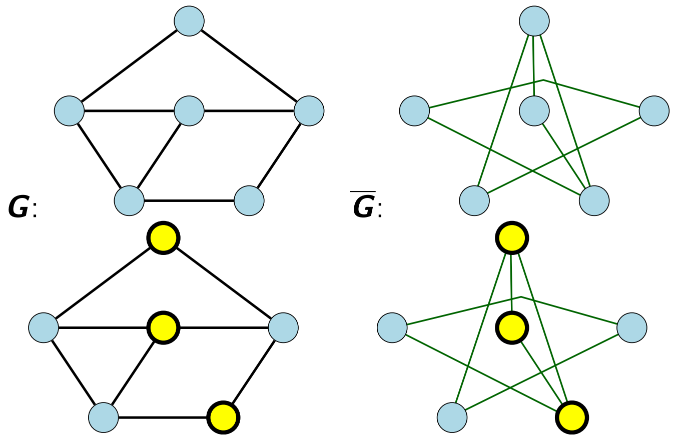
Types of Problems

Decision, Search, and Optimization

1. ***Decision problem***. Example: given n, is n prime?.
2. ***Search problem***. Example: given n, find a factor of n if it exists.
3. ***Optimization problem***. Example: find the smallest prime factor of n.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph G and an integer k. Convert G to $\overline{G}$, in which (u, v) is an edge $\iff$ (u, v) is not an edge of G. ($\overline{G}$ is the *complement* of G.)
$(\overline{G}, k)$: instance of **Clique**.



Independent Set and **Clique**

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.
4. Also... **Independent Set** is *at least as hard as* **Clique**.

Reductions, revised.

For decision problems X, Y, a ***reduction from X to Y*** is:

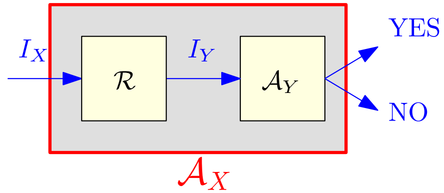
1. An algorithm ...
2. Input: I_X, an instance of X.
3. Output: I_Y an instance of Y.
4. Such that:
   $\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$

(Actually, this is only one type of reduction, but this is the one we'll use most often.)

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $X \to Y$
2. $\mathcal{A}_Y$: algorithm for Y:
3. $\implies$ New algorithm for X:

```
A_X(I_X):
        // I_X: instance of X.
        I_Y <= R(I_X)
        return A_Y(I_Y)
```

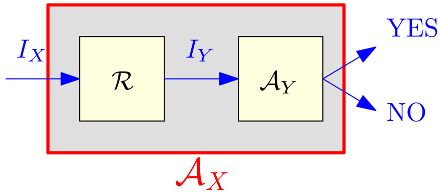


In particular, if $\mathcal{R}$ and $\mathcal{A}_Y$ are polynomial-time algorithms, $\mathcal{A}_X$ is also polynomial-time.

Comparing Problems

1. Reductions allow us to formalize the notion of "Problem X is no harder to solve than Problem Y".
2. If Problem X reduces to Problem Y (we write $X \leq Y$), then X cannot be harder to solve than Y.
3. More generally, if $X \leq Y$, we can say that X is no harder than Y, or Y is at least as hard as X.

Polynomial-time reductions



1. Algorithm is ***efficient*** if it runs in polynomial-time.
2. Interested only in polynomial-time reductions.
3. $X \leq_P Y$: Have polynomial-time reduction from problem X to problem Y.
4. $\mathcal{A}_Y$: poly-time algorithm for Y.
5. $\implies$ Polynomial-time/efficient algorithm for X.

Polynomial-time reductions and hardness

Lemma

For decision problems X and Y, if $X \leq_P Y$, and Y has an efficient algorithm, X has an efficient algorithm.

1. **Independent Set**: "believe" there is no efficient algorithm.
2. What about **Clique**?
3. Showed: **Independent Set** $\leq_P$ **Clique**.
4. If **Clique** had an efficient algorithm, so would **Independent Set**!

Observation

If $X \leq_P Y$ and X does not have an efficient algorithm, Y cannot have an efficient algorithm!

Part V

Polynomial time reductions.

Polynomial-time reductions and instance sizes

Proposition

$\mathcal{R}$: a polynomial-time reduction from X to Y. Then, for any instance I_X of X, the size of the instance I_Y of Y produced from I_X by $\mathcal{R}$ is polynomial in the size of I_X.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input I_X of size $|I_X|$ it runs in time $p(|I_X|)$ for some polynomial $p()$.
I_Y is the output of $\mathcal{R}$ on input I_X.
$\mathcal{R}$ can write at most $p(|I_X|)$ bits and hence $|I_Y| \leq p(|I_X|)$. □

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

Polynomial-time Reduction

Definition

A ***polynomial time reduction*** from a *decision* problem X to a *decision* problem Y is an *algorithm* $\mathcal{A}$ such that:

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. This implies that $|I_Y|$ (size of I_Y) is polynomial in $|I_X|$.
3. Answer to I_X YES $\iff$ answer to I_Y is YES.

Proposition

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

This is a ***Karp reduction***.

Transitivity of Reductions

Proposition

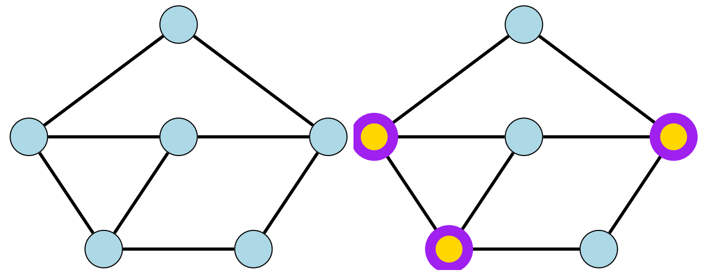
$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

1. Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.
2. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
3. ...show that an algorithm for Y implies an algorithm for X.

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A ***vertex cover*** if every $e \in E$ has at least one endpoint in S.



The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph* **G** *and integer* k.
Goal: *Is there a vertex cover of size* $\leq k$ *in* **G**?

Can we relate **Independent Set** and **Vertex Cover**?

Relationship between...

Vertex Cover and Independent Set

Proposition

Let $G = (V, E)$ *be a graph.*
S *is an independent set* $\iff$ $V \setminus S$ *is a vertex cover.*

Proof.

($\Rightarrow$) Let S be an independent set

0.1 Consider any edge $uv \in E$.
0.2 Since S is an independent set, either $u \notin S$ or $v \notin S$.
0.3 Thus, either $u \in V \setminus S$ or $v \in V \setminus S$.
0.4 $V \setminus S$ is a vertex cover.

($\Leftarrow$) Let $V \setminus S$ be some vertex cover:

0.1 Consider $u, v \in S$
0.2 uv is not an edge of **G**, as otherwise $V \setminus S$ does not cover uv.
0.3 $\implies$ S is thus an independent set. □

Independent Set $\leq_P$ **Vertex Cover**

1. G: graph with n vertices, and an integer k be an instance of the **Independent Set** problem.
2. G has an independent set of size $\geq k$ $\iff$ G has a vertex cover of size $\leq n - k$
3. (G, k) is an instance of **Independent Set** , and $(G, n - k)$ is an instance of **Vertex Cover** with the same answer.
4. Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

The **Set Cover** Problem

Problem (**Set Cover**)

Input: *Given a set* U *of* n *elements, a collection* $S_1, S_2, \ldots S_m$ *of subsets of* U, *and an integer* k.
Goal: *Is there a collection of at most* k *of these sets* S_i *whose union is equal to* U?

Example

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

$$S_1 = \{3, 7\} \quad S_2 = \{3, 4, 5\}$$
$$S_3 = \{1\} \quad S_4 = \{2, 4\}$$
$$S_5 = \{5\} \quad S_6 = \{1, 2, 6, 7\}$$

$\{S_2, S_6\}$ is a set cover

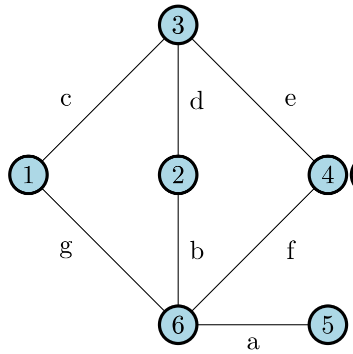
Vertex Cover $\leq_P$ Set Cover

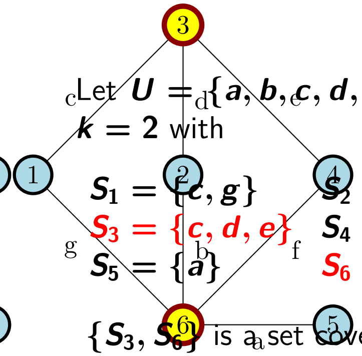
Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
2. $U = E$.
3. We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$.

Observe that G has vertex cover of size k if and only if $U, \{S_v\}_{v \in V}$ has a set cover of size k. (Exercise: Prove this.)

Vertex Cover $\leq_P$ Set Cover: Example





Let $U = \{a, b, c, d, e, f, g\}$, $k = 2$ with

$S_1 = \{c, g\}$ $S_2 = \{b, d\}$

$S_3 = \{c, d, e\}$ $S_4 = \{e, f\}$

$S_5 = \{a\}$ $S_6 = \{a, b, f, g\}$

$\{S_3, S_6\}$ is a set cover

$\{3, 6\}$ is a vertex cover

Proving Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES $\iff$ I_Y is YES.
   2.1 typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2.2 typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. Runs in *polynomial* time.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $X \leq_P Y$, and we have an efficient algorithm for Y, we have an efficient algorithm for X.
2. If $X \leq_P Y$, and there is no efficient algorithm for X, there is no efficient algorithm for Y.

We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.