CS 473: Algorithms

Sariel Har-Peled
sariel@illinois.edu
SC 1404

University of Illinois, Urbana-Champaign

Fall 2021

Administrivia, Introduction

Lecture 1

Tuesday, August 24, 2021

LaTeXed: August 26, 2021 12:26

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. Co-instructor: Bhaskar Ray Chaudhury
3. Teaching Assistants:
   1. Pooja Kulkarni
4. https://courses.engr.illinois.edu/cs473/fa2021/
5. Office hours: See course webpage
6. Email: See course webpage
7. **Tools**: Campuswire, gradescope, zoom.

Online resources

1. Webpage:
   https://courses.engr.illinois.edu/cs473/fa2021/
   General information, homeworks, etc.
2. Online questions/announcements: Piazza
   Online discussions, etc.
3. Gradescope: Submission of homeworks.

Textbooks

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

Recorded lectures from previous semester

Lectures of previous course are pre-recorded in small chunks. Might be useful in reviewing stuff...

https://courses.engr.illinois.edu/cs374/fa2020/lec_prerec/

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. Homeworks: 20%.
2. 2 Midterm(s): 25% each.
3. Final: 30% (covers the full course content).

Homeworks

1. One homework every week.
2. Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
3. Purpose of homeworks to prepare you for the exams.

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. Don't skip homework and don't copy homework solutions.
3. Study regularly and keep up with the course.
4. Ask for help promptly. Make use of office hours.

Homeworks

1. Homework 1 is posted on the class website. Quiz 0 available

Due warning

1. Challenging class.
2. Material is difficult.
3. Too much material, too little time.
4. Feel dazed and confused.

Part II

Course Goals and Overview

What we want

1. Modeling.
2. Algorithmic problem solving/thinking.
3. Reductions.
4. Know that you don't know.

Some problems...

1. "There are 125 sheep and 5 dogs in a flock. How old is the shepherd?"
2. There are 25 horses, every horse every time run the track in the same speed. But you can compare horses only if they run in the same race. A race can accommodate up to 5 horses. Design a tournament with min # of races such that you know who is the fastest horse.
3. Same question, but... Sort all the horses!

Topics

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

Goals

1. # Algorithmic thinking
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

What is an algorithm?

Subset Sum as integer programming

Input: $I = \{s_1, \ldots, s_n\}, t$: Positive integer numbers.
Q: Is there a subset $S \subseteq I$ of the numbers, such that

$$\sum_{s \in S} s = t.$$

Can be written as an integer program:

$$\begin{aligned} &\sum_{i=1}^{n} x_i s_i = t \\ &x_i \in \{0, 1\} && \forall i. \end{aligned}$$

Can one compute a solution?

Subset Sum as linear programming?

$$\sum_{i=1}^{n} x_i s_i = t$$

$$x_i \in \{0, 1\} \qquad \text{for } i = 1, \ldots, n.$$

Linear program:

$$\sum_{i=1}^{n} y_i s_i \leq t$$

$$\sum_{i=1}^{n} y_i s_i \geq t$$

$$0 \leq y_i \leq 1 \qquad \text{for } i = 1, \ldots, n.$$

Halting problem

Halting problem: Given a program $\boldsymbol{P}$ and an input $\boldsymbol{I}$, can one decide (i.e., always stop) if $\boldsymbol{P}$ stops on $\boldsymbol{I}$?

Turing: There is no program that can solve the halting problem.

The search space is unbounded as size of the input.

Part IV

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

1. How many bits to represent N in binary? $\lceil \log N \rceil$ bits.
2. Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.
   Exponential in the input size $n = \log N$.
3. 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:

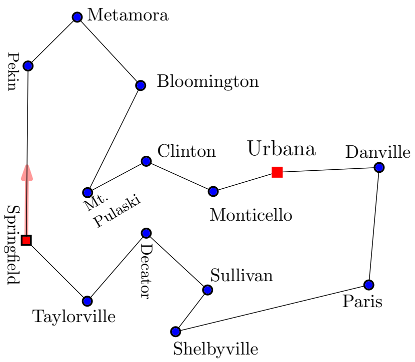
Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

Efficient algorithms

1. Is there an *efficient/good/effective* algorithm for primality?
2. **Question:** What does efficiency mean?
3. Here: ***efficiency*** is broadly equated to *polynomial time*.
4. $O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where n is size of the input.
5. Why? Is n^{100} really efficient/practical? Etc.
6. Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

TSP 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.
    1. Very close to optimal tour.
    2. Might have been optimal at the time..

Solving TSP 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 TSP by a Computer

Fastest computer...

1. Fastest super computer can do (roughly)

   $$\mathbf{2.5 * 10^{15}}$$

   operations a second.
2. Assume: computer checks $\mathbf{2.5 * 10^{15}}$ solutions every second, then...
   1. $\boldsymbol{n = 20} \implies$ 2 hours.
   2. $\boldsymbol{n = 25} \implies$ 200 years.
   3. $\boldsymbol{n = 37} \implies$ $\mathbf{2 * 10^{20}}$ years!!!

What is a good algorithm?

Running time...

"No, Thursday's out. How about never—is never good for you?"

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	$\mathbf{3 \cdot 10^9}$ 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
$\mathbf{10^8}$	4 secs	12.6839 years	$\mathbf{10^9}$ years	never
$\mathbf{10^9}$	6 mins	12683.9 years	$\mathbf{10^{13}}$ years	never

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 $\boldsymbol{A[i]}$)
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 V

Reductions

1.3: Independent Set and Clique

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\boldsymbol{V'}$ is:

1. ***independent set***: no two vertices of $\boldsymbol{V'}$ connected by an edge.

Independent Sets and Cliques

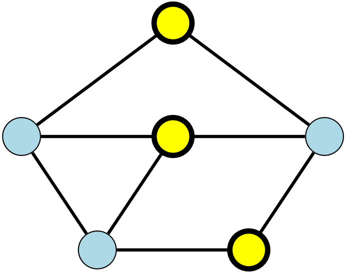
Given a graph $\mathbf{G}$, a set of vertices V' is:

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

Independent Sets and Cliques

Given a graph **G**, a set of vertices ***V′*** is:

1. ***independent set***: no two vertices of ***V′*** connected by an edge.



Independent Sets and Cliques

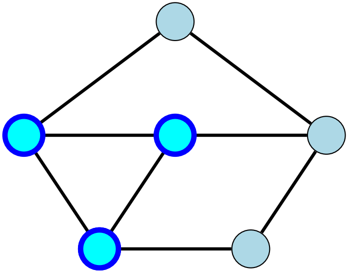
Given a graph $\mathbf{G}$, a set of vertices $\boldsymbol{V'}$ is:

1. ***independent set***: no two vertices of $\boldsymbol{V'}$ connected by an edge.
2. ***clique***: *every* pair of vertices in $\boldsymbol{V'}$ connected by an edge of $\mathbf{G}$.

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\boldsymbol{V'}$ is:

1. ***independent set***: no two vertices of $\boldsymbol{V'}$ connected by an edge.
2. ***clique***: *every* pair of vertices in $\boldsymbol{V'}$ connected by an edge of $\mathbf{G}$.



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

Problem: **Independent Set**

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

Problem: **Clique**

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

Types of Problems

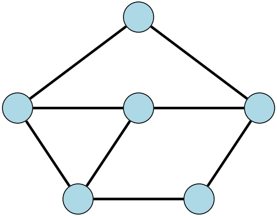
Decision, Search, and Optimization

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***.

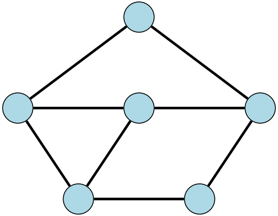
G:



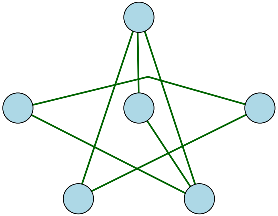
Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph **G** and an integer k.

G:

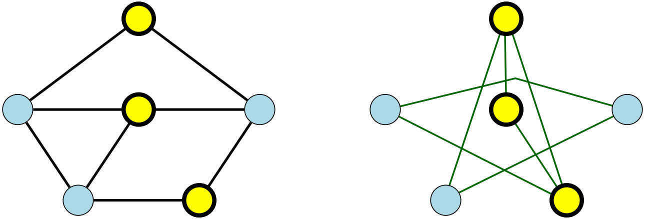


$\overline{G}$:



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

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

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

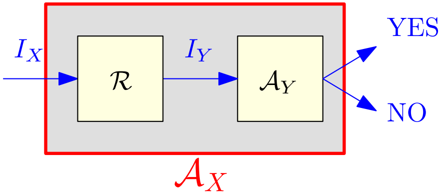
$$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$

There are other kinds of reductions.

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:

$\mathcal{A}_X(I_X)$:
    `//` I_X`: instance of` X.
    $I_Y \Leftarrow \mathcal{R}(I_X)$
    return $\mathcal{A}_Y(I_Y)$

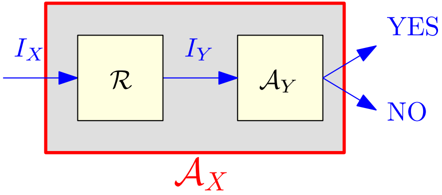


If $\mathcal{R}$ and $\mathcal{A}_Y$ polynomial-time $\implies$ $\mathcal{A}_X$ polynomial-time.

Comparing Problems

1. "Problem $\boldsymbol{X}$ is no harder to solve than Problem $\boldsymbol{Y}$".
2. If Problem $\boldsymbol{X}$ reduces to Problem $\boldsymbol{Y}$ (we write $\boldsymbol{X} \leq \boldsymbol{Y}$), then $\boldsymbol{X}$ cannot be harder to solve than $\boldsymbol{Y}$.
3. $\boldsymbol{X} \leq \boldsymbol{Y}$:
    1. $\boldsymbol{X}$ is no harder than $\boldsymbol{Y}$, or
    2. $\boldsymbol{Y}$ is at least as hard as $\boldsymbol{X}$.

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 $\boldsymbol{X}$ and $\boldsymbol{Y}$, if $\boldsymbol{X} \leq_{\boldsymbol{P}} \boldsymbol{Y}$, and $\boldsymbol{Y}$ has an efficient algorithm, $\boldsymbol{X}$ has an efficient algorithm.

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

Observation

If $\boldsymbol{X} \leq_{\boldsymbol{P}} \boldsymbol{Y}$ and $\boldsymbol{X}$ does not have an efficient algorithm, $\boldsymbol{Y}$ cannot have an efficient algorithm!