CSE347 Class 1

Jeremy Buhler

August 28, 2017

`https://classes.engineering.wustl.edu/cse347/`

1 Welcome and Administrivia

Welcome to CSE 347!

- This is a course about algorithms and their analysis.
- We'll see a variety of paradigms for how to design algorithms, and a variety of techniques for understanding why they are correct and how efficient they are.
- Eventually, we'll study *intractability* – why some computational problems are unlikely to have an efficient algorithm, and what to do about it.
- "Active learning" – no good unless you engage with the material!!!

How is this course structured?

- **Class Meetings**: fundamental methods and ideas
- **Recitation**: led by senior TAs, to develop proof skills in an interactive setting and improve understanding of material
- **Homeworks** (45%): graded problems to make you work with the material creatively. Roughly one per week.
- **Exams** (55%): final opportunity to demonstrate that you've mastered the material. One midterm and one final.

What do you need to know right now?

- The course web site is your friend. All assignments appear there.
- Course discussions and communication with instructors and TAs is conducted through our Piazza board.
- Homeworks are submitted electronically via Blackboard. See the Electronic Homework Submission Guide on the website for detailed instructions and help with formatting.
- Read the course overview! It has all the grody administrative details.

- There is a **collaboration policy** for homeworks – tries to balance need for collective problem solving with promotion of individual skill building and assessment.
  - You can seek help from others and outside sources in working on problems.
  - All final writeups must be in your own words.
  - You must attribute any outside help you received when turning in your homework.

Who is here to help you, and how do you contact us?

- Lots of TAs, including three recitation instructors. Listed on Piazza.
- Our office hours are all posted on Piazza.
- To contact myself or TAs (or to initiate a discussion with your friends), use Piazza.

What do I expect you to know already?

- First: how to write a very basic formal proof.
- But we'll also learn proof strategies specific to the material.
- Second: asymptotic complexity analysis (for few algorithmic components)
- To warm up on formal proofs, there is an ungraded "Diagnostic" on the web site. You should be able to do all the problems found there and produce suitable proofs.
- You can also use the diagnostic to make sure you know how to submit homework problems through Blackboard.
- If you have doubts about the Diagnostic or want us to look at it, please submit it no later than September 5 (at 11:59 PM).

OK, on to the fun stuff...

2 Why Are We Here?

We're here to study algorithms and their analysis.

- An "algorithm" is an "effective procedure for solving a computational problem."
- "Effective" means you can implement it on a computer (for various equivalent definitions of "computer.")
- We generally want to know two things about an algorithm
  1. Why does it work (correctness)?
  2. How fast is it (efficiency)?
- To get started, we'll rewind to the earliest *well-known* example of an algorithm, from the ancient Greeks (no promises – who knows what the Babylonians had?).

- Suppose you have integers $a > b$.
- The *greatest common divisor* of a and b, denoted $\text{GCD}(a, b)$, is the largest integer d such that d divides both a and b.
- That is, $a = xd$ and $b = yd$ for some $x, y > 0$.
- **Example**: $\text{GCD}(100, 35)$?
- Well, $35 = 5 \cdot 7$, and $100 = 5^2 \cdot 2^2$, so their GCD is 5.
- You probably learned about GCD years ago in school.
- **Problem**: given $a > b$, how can we compute $\text{GCD}(a, b)$?
- If you had the same grade-school curriculum as my daughter, you probably learned to do *trial division*: divide by primes 2, 3, 5, etc. until you find p that divides both a and b, then repeat on a/p and b/p until there is no common divisor $< b$.
- This is clearly correct – we'll eventually find all common prime factors.
- But is it efficient?
- Well, there are $O(x/\log x)$ prime numbers less than x, so one round of trial division takes $O(b/\log b)$ divisions.
- Since a number b has $\Theta(\log b)$ digits, one round of trial division requires a number of divisions that is worst-case superpolynomial in the *number of digits* in b.
- Finding all factors takes multiple rounds of trial division.

Is there a more efficient, equally correct algorithm for computing GCD?

3 Euclid's Algorithm

- Let's study an algorithm for GCD that is about 2300 years old.
- It was published in Euclid's *Elements*.
- If any divisor d divides both of $a > b$, then d surely divides $a - b$.
- Hence, $\text{GCD}(a, b) = \text{GCD}(b, a - b)$.
- In fact, d must also divide $a - 2b$, $a - 3b$, and so forth until we can't subtract any more b's.
- To save steps, we can replace all the subtractions by a modulo operation (equivalent to one division):
$$\text{GCD}(a, b) = \text{GCD}(b, a \bmod b).$$
- Now we still have a GCD problem left, but one of the numbers is smaller, so we've made progress.

- With this step in mind, we give the full algorithm:

```
EUCLID(a,b)                          ▷ a > b ≥ 0
  (r, s) ← (a, b)
  while s > 0 do
      (r, s) ← (s, r mod s)
  return r
```

- **Example**: GCD$(14191, 12957)$

$$\begin{aligned} 14191 &= 1 \cdot 12957 + 1234 \\ 12957 &= 10 \cdot 1234 + 617 \\ 1234 &= 2 \cdot 617 + 0 \end{aligned}$$

- By our algorithm, the desired GCD is 617. And indeed, $14191 = 23 \cdot 617$, while $12957 = 6 \cdot 617$, and 23 and 6 are relatively prime.

Does this algorithm always work? Is it fast?

4 Analysis of Euclid's Algorithm

- **Claim**: Euclid's algorithm always returns the correct GCD.
- **Pf**: by induction on # of iterations of the while loop.
- If the loop runs zero times, then the smaller of r, s is 0.
- But r always divides 0 (in particular, $0 \cdot r = 0$), so the algorithm returns correct the GCD in this case, which is $r = a$.
- If the loop runs more than zero times, we argued above that after each step, the GCD of the two values r, s remains the same.
- Moreover, the loop always makes progress, because the larger of the two values decreases due to the mod operation.
- Conclude that in finite time, the algorithm will terminate, and the last step returns the correct GCD by our argument for the base case. QED

Yay, the algorithm works! But how can we quantify how fast it is?

- We'll quantify the cost in terms of the number of division/modulus steps, i.e. the number of iterations of the while loop.
- **Thm** (Lamé, 1844): Euclid's algorithm on inputs $a > b$ requires at most $\frac{\log b}{\log \phi} + 1$ iterations.
- Here, $\phi = (1 + \sqrt{5})/2$ is the "golden ratio." It has the important property (for this proof) that $\phi + 1 = \phi^2$.

- **Pf**: We claim that, if the algorithm needs to run for i iterations to complete, then $r \geq \phi^i$.
- (This claim implies that if the algorithm runs for a total of ℓ iterations on (a, b), then b, which becomes r after one iteration, is $\geq \phi^{\ell-1}$. Solving for ℓ in terms of b yields the statement of the theorem.)
- Proof is by induction on i, the number of iterations remaining.
- **Bas 1**: when $i = 0$, $r > s = 0$, so $r \geq 1 = \phi^0$.
- **Bas 2**: when $i = 1$, $r > s > 0$, so $r \geq 2 \geq \phi^1$.
- **Ind**: when $i \geq 2$, we know that
  - after *one* more iteration, r will equal current iteration's value of s; hence, by IH, $s \geq \phi^{i-1}$;
  - after *two* more iterations, r will equal current iteration's r mod s; hence, by IH, r mod $s \geq \phi^{i-2}$.
- But $r \geq (r \bmod s) + s$, since $r \geq s$ and so $r = xs + (r \bmod s)$ for some $x > 0$.
- Conclude that

$$\begin{aligned} r &\geq \phi^{i-1} + \phi^{i-2} \\ &= \phi^{i-2}(\phi + 1) \\ &= \phi^{i-2}\phi^2 \\ &= \phi^i. \end{aligned}$$

  .
- **Cor**: Euclid's algorithm performs $O(\log b)$ divisions, which is linear in the number of digits in b.

5 Extending Euclid's Algorithm

- Suppose we want to solve an equation of the form $a \cdot x + b \cdot y = d$, for fixed a, b, d.
- Assume that a, b, d are integers, and we *only* want solution values x, y that are themselves integers.
- This is called a *linear Diophantine equation.*
- Such an equation may or may not have a solution.
- **Ex**: $6x + 8y = 7$ has no solutions, because any integer choices of x, y yield an even integer value.
- **Ex**: $6x + 8y = 12$ has many solutions, e.g.: $x = 2, y = 0$; $x = -6, y = 6$; $x = -10, y = 9$; $x = 14, y = -9$; and so forth.
- Now suppose in particular that the RHS $d = \text{GCD}(a, b)$; can we mechanically find a solution (x, y)?

- To formalize a bit,

  > Given integers a and b with $d = \text{GCD}(a, b)$, is there an algorithm to compute x, y such that $ax + by = d$?

We can leverage Euclid's algorithm to solve this probelm!

- Let r_j be the value of r generated in Euclid's algorithm on input (a, b) after the ith iteration.
- In particular, $r_0 = a$ and $r_1 = b$.
- In the ith iteration, the division/modulo operation divides r_{j-1} into a multiple $q_j r_j$ and a remainder $r_{j-1} \bmod r_j$, s.t.

$$r_{j-1} = q_j r_j + (r_{j-1} \bmod r_j).$$

- (The pseudocode shows us computing only the remainder, but we could compute the quotient q_j too if we wanted.)
- But in fact, $r_{j-1} \bmod r_j$ becomes r_{j+1} in a later iteration, so we have

$$r_{j-1} = q_j r_j + r_{j+1},$$

  or equivalently,

$$r_{j+1} = r_{j-1} - q_j r_j.$$

- When the algorithm finishes, $r_\ell = d$, and we have computed all the q_j values.
- We can now start with the final equation

$$d = r_{\ell-2} - q_{\ell-1} r_{\ell-1}$$

  and repeatedly back-substitute to remove the largest r_j from the RHS.

- For example, in one step, we can rewrite as

$$\begin{aligned} d &= r_{\ell-2} - q_{\ell-1}(r_{\ell-3} - q_{\ell-2} r_{\ell-2}) \\ &= (1 - q_{\ell-1} q_{\ell-2}) r_{\ell-2} - q_{\ell-1} r_{\ell-3} \end{aligned}$$

  to remove the $r_{\ell-1}$ term, leaving d as a linear combination of $r_{\ell-2}$ and $r_{\ell-3}$.

- Repeating this process, we eventually derive

$$\begin{aligned} d &= x r_1 + y r_0 \\ &= xb + ya \end{aligned}$$

  for some computed x and y.

- This takes only $O(\log b)$ total arithmetic operations.
- This back-substitution process yields the *extended Euclid algorithm*.

Extended Euclid can solve linear Diophantine equations of the form

$$ax + by = \gcd(a, b).$$

What is this good for?

- Consider arithmetic modulo a number p.
- We can always add, subtract, and multiply numbers modulo p.
- If p is prime, we can also *divide* modulo p.
- In other words, for every $a < p$ (and in general, for a not a multiple of p), *there exists* a unique integer $b < p$ s.t. $ab = 1 \mod p$.
- (This is a theorem from number theory that I'm not going to prove.)
- Hence, it makes sense to write "$b = 1/a \pmod p$" or equivalently "b is the multiplicative inverse of a, modulo p."
- But how do we actually *compute* the multiplicative inverse of a number mod p?
- Note that $\mathrm{GCD}(a, p) = 1$. If we use extended Euclid to solve the equation

  $$xa + yp = 1$$

  in integers, the (unique) value of x between 1 and $p - 1$ that solves the equation is our desired b.
- (If the algorithm gives us an x not in this range, we can simply take that result modulo p to get the value we want.)
- Note that we can in general perform a division operation $c/d \bmod p$ by computing $c \cdot 1/d \bmod p$.

Here's another application, for those interested...

- We can now efficiently add, subtract, multiply, *and* divide modulo a prime p.
- We are used to solving linear systems of equations $A \cdot x = b$ in real numbers.
- Algorithms for finding such solutions (generally in real or rational numbers) entail applying some sequence of the four arithmetic operations.
- We can equally well implement such algorithms to solve linear systems over the integers modulo p, because the four operations are all well-defined and (as we've now seen) efficiently computable.
- We'll use the ability to solve linear systems modulo a prime when we talk about universal hashing.

And your homework will show yet a third example.