Computational Discrete Mathematics: Handout 03

Curtis Bright

September 16, 2021

1 The Euclidean Algorithm

This worksheet provides an introduction to the Euclidean algorithm—in its most basic form, a way to find the largest possible number that evenly divides two other numbers.

Knuth says the Euclidean algorithm is "the oldest nontrivial algorithm that has survived to the present day".

The algorithm has a huge number of applications, from cryptography to coding theory to solving linear equations over the integers.

1.1 Greatest Common Divisor

The *greatest common divisor* (gcd) of two integers is the largest number that "divides" each of them. For example, the greatest common divisor of 15 and 24 is 3.

The (positive) divisors of 15 are 1, 3, 5, and 15. The (positive) divisors of 24 are 1, 2, 3, 4, 6, 8, 12, 24. These can be computed with the `divisors` command:

[ ]:
```
show(divisors(15))
show(divisors(24))
```

The `gcd` command computes the gcd of two numbers:

[ ]:
```
show(gcd(15, 24))
```

We can also compute the gcd of two polynomials:

[ ]:
```
R.<x> = ZZ[]
a = x^4-x^3-3*x^2+x+2
b = x^3-4*x^2+x+6
g = gcd(a, b)
show(g)
```

To double-check this computation, we could also explicitly factor the polynomials and look for their common factors:

[ ]:
```
show(factor(a))
show(factor(b))
show(factor(g))
```

1.2 Computation of gcd

How should the gcd of two numbers (or polynomials) be computed? The most straightforward way of finding all the divisors of the numbers and then selecting the greatest is quite inefficient and not a good way except in small cases.

Fortunately, the Euclidean algorithm provides a different and efficient way of computing greatest common divisors.

It is based on the following simple identity:

$$\gcd(a, b) = \gcd(a, b - a)$$

This holds as a result of the property that any number which divides both a and b also divides their difference.

1.2.1 How does this help?

At first this might look useless, since it would seem that we would need to already know how to compute gcds in order to make use of it.

However, it suggests a recursive algorithm. Suppose $b > a > 0$ so that $b - a$ is positive but smaller than b. In this sense computing $\gcd(a, b - a)$ is simpler than computing $\gcd(a, b)$.

Example: $\gcd(15, 24) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 3) = \gcd(3, 0) = 3$

Repeatedly subtracting the larger from the smaller number must eventually reach 0 and we know that $\gcd(x, 0) = x$.

1.2.2 A closer look

Repeated subtractions are wasteful. For example:

$$\gcd(1, 1000) = \gcd(1, 999) = \gcd(1, 998) = \cdots = \gcd(1, 1) = \gcd(1, 0) = 1$$

A better idea: subtract off as many multiples of a as possible at each step. In other words, we use the identity

$$\gcd(a, b) = \gcd(a, b - qa)$$

where q is as large as possible so that $b - qa$ remains positive.

Look familiar? The largest value of q for which $b - qa$ is positive is $q = \lfloor b/a \rfloor$ (the *floor* of b/a).

Then $b - qa$ is just the remainder of b divided by a (commonly denoted by b mod a). In the worksheet on basic algebraic operations we already saw how to compute this.

1.3 The Basic Euclidean Algorithm

Here's basic pseudocode for the Euclidean algorithm on positive $b \geq a$:

```
if a = 0 then
    return b
# Compute remainder of b divided by a
r = b % a
return gcd(r, a)
```

In fact, this even works if $a > b$, since then r will be set to b and $\gcd(b, a)$ will be returned—in other words, the order of the parameters will be swapped so the first parameter is larger.

1.3.1 Sage implementation

[ ]:
```
# Compute the gcd of a and b
def my_gcd(a, b):
    if a == 0:
        return b
    return my_gcd(b % a, a)

my_gcd(15, 24)
```

1.4 The Iterative Euclidean Algorithm

The above algorithm can also be written iteratively to avoid recursion.

We start by setting $r_0 := a$, $r_1 := b$ and then iteratively compute new remainders via

$$r_{i+1} := r_{i-1} \bmod r_i.$$

It follows that $\gcd(r_0, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_l, r_{l+1})$ and once a remainder in this sequence becomes 0 we can stop and return the last nonzero entry.

1.4.1 Sage implementation

[ ]:
```
# Compute the sequence of remainders in the Euclidean algorithm on (a, b)
def remainder_sequence(a, b):
    r = [a, b]
    while r[-1] != 0:
        r.append(r[-2] % r[-1])
    return r

show(remainder_sequence(15, 24))
```

1.4.2 Big examples

The beauty of this algorithm is that works well on very large numbers.

[ ]:
```
# Compute the remainder sequence of two primes with 50 bits
show(remainder_sequence(random_prime(2^50), random_prime(2^50)))
```

[ ]:
```
a = (2^200).next_prime()
b = (2*a).next_prime()
c = (2*b).next_prime()
show([a, b, c])
```

[ ]:
```
my_gcd(a*b, b*c)
```

1.4.3 A polynomial example

The same basic algorithm works for polynomials as well. Recall that in this case the "size" of a polynomial is given by its degree. In this case, the remainders will form a sequence that strictly decrease in degree.

Note that the gcd is not unique here: if g is a gcd then any constant multiple of g will also be a gcd. To have a single canonical answer, we can enforce uniqueness by specifying that the leading coefficient of a gcd is 1.

[ ]:
```
# Compute with polynomials with rational coefficients
R.<x> = QQ[]
a = x^4-x^3-3*x^2+x+2
b = x^3-4*x^2+x+6

# Compute remainder sequence of a and b in R = QQ[x]
def remainder_sequence_poly(a, b):
    r = [a, b]
    while r[-1] != 0:
        g = r[-2] % r[-1]
        if g != 0:
            g = g/g.leading_coefficient()
        r.append(g)
    return r

# Compute normalized gcd of a and b
def my_gcd_poly(a, b):
    g = remainder_sequence_poly(a, b)[-2]
    return g/g.leading_coefficient()

show(a)
show(b)
show(remainder_sequence_poly(a, b))
show(my_gcd_poly(a, b))
```

[ ]:
```
# A larger example - note that the coefficients of the remainder polynomials␣
↪may not be integers
```

```
a = (x+1)^2*(x-1)^5
b = (x+1)^5*(x-1)^2
show(remainder_sequence_poly(a, b))
show(my_gcd_poly(a, b))
```

[ ]:
```
show(my_gcd_poly(a, b) == (x+1)^2*(x-1)^2)
```

1.5 Algorithm Analysis

We saw that Euclid's algorithm seems fast, but *how* fast?

Suppose that a and b have length at most n. On each iteration the remainders decrease so all remainders have length at most n. Thus the computation of each remainder uses $O(n^2)$ word operations. Since the remainders decrease on each loop iteration there can be at most n iterations.

Thus, Euclid's algorithm on length n integers requires $O(n^3)$ word operations. Similarly, Euclid's algorithm on degree n polynomials requires $O(n^3)$ coefficient operations.

While this is correct, it is not actually a tight bound: the analysis can be improved. Recall from the previous worksheet that division with remainder on operands of size n and m requires $O((n - m + 1) \cdot m)$ word/coefficient operations. Say that c is a constant hidden by the O notation, i.e., the remainder can be performed in at most $c \cdot (n - m + 1) \cdot m$ operations.

Let n_i denote the size of r_i and say there are up to l iterations of the while loop. The total number of word/coefficient operations used is at most $\sum_{i=1}^{l} c \cdot (n_{i-1} - n_i + 1) \cdot n_i$.

Note that $\sum_{i=1}^{l}(n_{i-1} - n_i + 1) = n_0 - n_l + l \leq 2 \cdot n$ since each term cancels with the subsequent term and there are at most n iterations of the loop.

Then the total cost is at most

$$\sum_{i=1}^{l} c \cdot (n_{i-1} - n_i + 1) \cdot n_i \leq c \cdot n \cdot \sum_{i=1}^{l}(n_{i-1} - n_i + 1) \leq 2 \cdot c \cdot n^2$$

operations. In other words, a quadratic cost of $O(n^2)$ operations.

1.6 The Extended Euclidean Algorithm

One of the many applications of the Euclidean algorithm is to solve linear equations over the integers—known as linear Diophantine equations.

For example, given integers a, b, and d can you solve

$$ax + by = d$$

for integers x and y?

1.6.1 Bézout's identity

Bézout's identity describes exactly when this equation has solutions—namely, exactly when d is a multiple of $\gcd(a, b)$. In particular, solutions must exist when $d = \gcd(a, b)$. The *Extended* Euclidean algorithm (EEA) gives a method that solves this equation.

The EEA performs the same operations as the normal Euclidean algorithm but it also keeps track of additional information. In particular, on the $(i-1)$th loop iteration this extra information is a solution pair (x, y) of the equation

$$ax + by = r_i. \tag{$*$}$$

Then the second-last iteration provides a solution of $ax + by = r_{l-1} = d$ (where there are l loop iterations).

1.6.2 Initializing the EEA

To start off with, note that we can easily find solutions of $(*)$ for the initial values $i = 0, 1$.

In particular, we have

$$\begin{aligned} a \cdot 1 + b \cdot 0 &= a = r_0 \\ a \cdot 0 + b \cdot 1 &= b = r_1. \end{aligned}$$

In other words, if (x_i, y_i) denotes a solution to $(*)$ then we can start off with

$$\begin{aligned} (x_0, y_0) &= (1, 0) \\ (x_1, y_1) &= (0, 1). \end{aligned}$$

1.6.3 Iteration step in the EEA

Suppose we are on the ith loop iteration of the EEA. On the previous two loop iterations we stored solutions (x_{i-2}, y_{i-2}) and (x_{i-1}, y_{i-1}) which satisfy

$$\begin{aligned} ax_{i-2} + by_{i-2} &= r_{i-2} \\ ax_{i-1} + by_{i-1} &= r_{i-1}. \end{aligned}$$

Recall that $r_i = r_{i-2} \bmod r_{i-1}$ since $r_i = r_{i-2} - qr_{i-1}$ where $q = \lfloor r_{i-2}/r_{i-1} \rfloor$. Note that we can take the first of the two equations above and subtract off q copies of the second equation to form the new equation

$$a(x_{i-2} - qx_{i-1}) + b(y_{i-2} - qy_{i-1}) = r_{i-2} - qr_{i-1} = r_i.$$

Thus, we set $x_i := x_{i-2} - qx_{i-1}$ and $y_i := y_{i-2} - qy_{i-1}$.

1.6.4 Example

The following is a Sage implementation of the EEA (along with printing the sequence of equations produced as output for demonstration). Note that // in sage denotes integer division, i.e., a//b means $\lfloor a/b \rfloor$.

[ ]:
```
# Returns integers (x,y) such that a*x+b*y = gcd(a,b)
def eea(a, b):
    r = [a, b]
    x = [1, 0]
    y = [0, 1]
    print("{}*({}) + {}*({}) = {}".format(a, x[0], b, y[0], r[0]))
    print("{}*({}) + {}*({}) = {}".format(a, x[1], b, y[1], r[1]))
    while r[-1] != 0:
        q = r[-2]//r[-1]
        r.append(r[-2]-q*r[-1])
        x.append(x[-2]-q*x[-1])
        y.append(y[-2]-q*y[-1])
        print("{}*{} + {}*{} = {}".format(a, x[-1], b, y[-1], r[-1]))
    return x[-2], y[-2]

a, b = 15, 24
x, y = eea(a, b)
show([x,y])
```

[ ]:
```
show(x*a + y*b)
```

1.6.5 Analysis of the EEA

The analysis of the EEA proceeds in the same basic way as the analysis of the standard Euclidean algorithm except now there are a few new expressions that add to the cost—in particular, the two new multiplications qx_{i-1} and qy_{i-1}.

This is slightly tricky to analyze as the values of x_i and y_i increase in absolute value as i increases. However, one can show that $|y_i| \leq a$ and $|x_i| \leq b$ will hold for all i. Thus y_i has length at most n and x_i has length at most m and these bounds will be sufficient in the analysis.

Let q denote the value of q on the ith iteration. One can show that $\text{len}(q_i) \leq \text{len}(r_{i-1}) - \text{len}(r_{i-2}) + 1$ and using this for $1 \leq i \leq l$ we derive

$$\sum_{i=1}^{l} \text{len}(q_i) \leq \text{len}(r_0) - \text{len}(r_l) + l \leq 2n$$

by a similar "term cancellation" argument as before.

The total cost of computing all multiplications $q_i \cdot y_i$ for $i = 1, \ldots, l$ is then

$$\sum_{i=1}^{l} \text{len}(y_i) \cdot \text{len}(q_i) \leq \text{len}(a) \cdot \sum_{i=1}^{l} \text{len}(q_i) \leq n \cdot (2n) = 2n^2.$$

The multiplications $q_i \cdot x_i$ are similarly handled and this gives a total running cost of $O(n^2)$ word operations to perform the extended Euclidean algorithm.

1.6.6 Built-in Sage function

Sage also provides the built-in function which performs the EEA: `xgcd(a,b)` returns a triple (g, x, y) satisfying $g = xa + yb$ where $g = \gcd(a, b)$.

[ ]:
```
xgcd(15,24)
```