1 GCD and Bezout's Identity

- *Algorithm:* Several Algorithms are shown below:
  Finding the GCD:
  Prime Factorization
  Naive Euclid's Algorithm(algo. 1)
  Binary Algorithm(algo. 2)
  Solving Bezout's Identity:
  Extended Euclidean Algorithm(algo. 3)
- *Input:* Two integers a and b, we identify that $a \leq b$.
- *Complexity:* $\mathcal{O}((\log n)^2)$
- *Data structure compatibility:* N/A
- *Common applications:* Can be also used to find the GCD for polynomials and the divisors for principle ideal domain. Also the extended Euclidean Algorithm can be used to find the multiplicative inverses which is now widely used in RSA (see algo. 4).

Problem. GCD and Bezout's Identity

GCD, representing the Greatest Common Divisor, is a problem with a long history. It is defined as the largest positive number that divides both a and b.

The naivest way of finding GCD is using prime factorization, but since prime factorization is $co - \mathcal{NP}$, it is really time-consuming and is only suitable for finding GCD for small numbers. Euclid raised another idea about how to calculate the GCD, which is named Euclid's Algorithm, which makes the problem simple for large numbers. After that, many other algorithm has been raised.

Bezout's identity is a new idea based on the GCD. It is that let d be the GCD of a and b, then we can find some integers x and y that satisfies $ax + by = d$. x and y here are called Bezout's coefficients, and they are not unique. In addition, the d is the smallest number n that can be expressed as $ax + by = n$, and every n here should be a multiple of d. The x and y here can be calculated by Extended Euclidean Algorithm.

Description

Detailed description of the problem; More detailed information on the input and complexity; more applications with details on how they relate to each other (if this is the case).

GCD Algorithms:

1. Prime Factorization:

GCD's naivest solution is to use the prime factorization. For two numbers a and b, we calculate the prime factorization of each of them, and we get a set of numbers. And then we intersection of them, then we multiple them together, which is the GCD.

However, we know that prime factorization is not a $\mathcal{P}$ question, so it really cost a lot of time to solve it if a and b are relatively large. So we only use it when the numbers are small.

2. Euclid's Algorithm:

Before talking about what Euclid's Algorithm is, let's first consider the $gcd(a, b) = d$. Given a rectangle which has a width a and a length b, and thus both the length and the height can be divided into small segments with length d, that is, the rectangle can be divided into small square grid with length d.

The Euclid's idea is based on this. Given the rectangle, suppose that we are required to divide the rectangle into the small grids given above. We should find the small grid. We first cut a square grid off with length $min\{a, b\}$, and since $d \mid min\{a, b\}$, we can say that the square grid can be filled with the small grid mentioned above. We repeat the procedure, and then we can get a grid remaining in the end, and the grid is exactly what we want.

If we represent the procedure into the form of algorithm, then we have some algorithms. Those algorithms includes division based, subtraction based and recursive based.

Algorithm 1: Euclid's Algorithm - division based

Input : Two positive integers a and b

Output: Their GCD

```
1 Function EuclidDivision(a, b):
2     while b ≠ 0 do
3         t ← b
4         b ← a mod b
5         a ← t
6     end while
7     return a
8 end
```

The subtraction based algorithm is similar, where we just subtract the small number from the larger number. The recursive based is similar as well, only changing the while loop into some recursions. Subtraction based algorithm is exactly the one based on the rectangle lemma, while the division based is just subtract all the square grids with same length at a time.

The time complexity of this algorithm is $\mathcal{O}(\log \max\{a, b\})$. From the previous paragraph, we can see that division based algorithm is much more better than the subtraction based algorithm, so the time needed for a division based is smaller than or equal to the subtraction based one. The worst case is that if a and b are some closest Fibonacci number, that is, $b \bmod a = b - a$, where we suppose $b > a$ here.

Let b be the i-th Fibonacci number, while a be the (i-1)-th. So when doing one while loop, the b and a will go down one step, so the total loops should be i. Also, the i-th Fibonacci number can be represented by $F_i = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2}^i - \frac{1-\sqrt{5}}{2}^i)$, where we can conclude that there exists some constant c that $c\log F_i \leq i$. So the time complexity is $\mathcal{O}(\log \max\{a, b\}) = \mathcal{O}(\log a + \log b)$. When adding the time complexity of calculating a **mod** b, we have the total computational time complexity $\mathcal{O}((\log n)^2)$.

The complete proof for why two closest Fibonacci number is the worst case is given by **Lame's Theorem**.

3. Binary Algorithm:

The binary algorithm is used to divide the GCD into the forms of $g \times 2^d$, where g is odd. The idea is that for some numbers a and b, if a is even while b is odd, then the common division of a and b must also be the

common division of $a/2$ and b. If both even, and g, which is odd, is a common division of a and b, then $2g$ is also a common division of a and b, that is g is a common division of $a/2$ and $b/2$.

The more complicated case is that if a and b are both odd, we can see that the GCD is also the GCD for $b - a$ where $b > a$. Then we can see that $b - a$ is an even number and we can do this process recursively.

Algorithm 2: Binary Algorithm

Input : Two positive integers a and b
Output: Their GCD

```
d ← 0
while a and b are both even do
    a ← a/2
    b ← b/2
    d ← d + 1
end while
while a ≠ b do
    if a is even then
        a ← a/2
    end if
    else if b is even then
        b ← b/2
    end if
    else if a > b then
        a ← (a − b)/2
    end if
    else
        b ← (b − a)/2
    end if
end while
g ← a
return g · 2^d
```

Each of the step cost exactly one shift of the while loop, so the total time complexity is similar to the one of the Euclid's Algorithm. The time complexity is $\mathcal{O}(\log a + \log b)$. In addition, the subtract of two numbers cost $\mathcal{O}(\log a + \log b)$, so the total time complexity is $\mathcal{O}((\log a + \log b)^2)$

Bezout's Identity Algorithms:

1. Proof for the Bezout Identity:

For some integers a and b, suppose the set $I = \{ax + by \mid x, y \in \mathbb{Z}\}$. Let $x, y = 1$ if the corresponding $a, b = 1$ and $x, y = -1$ if the corresponding $a, b = -1$, we can find an element $ax + by \in I$ and $ax + by \leq 0$.

Suppose $I \cap \mathbb{N} = P$. So $P = \emptyset$, so P must have a smallest element. Suppose the smallest element is d.

Since $d \in I$, so there exists some x and y that $ax + by = d$. Also, we have $a = dq + r$, where q and r are integers and $r \in [0, d)$. Suppose $r > 0$, then we have

$$r = a - dq = a - (ax + by)q = a(1 - xq) + b(-yq)$$

So r is also in I, which is contradict to d is the smallest element in P. So $r = 0$, which means that d is a division of a. Similarly, d is also a division of b. Then we have d is a common division of a and b. Let c be any common division of a and b. Then let $a = ca'$ and $b = cb'$. So $d = c(a'x + b'y)$, so $c \mid d$, so d is the greatest common division, i.e. $d = gcd(a, b)$.

2. Extended Euclidean Algorithm:

Then we developed the Extended Euclidean Algorithm to find the coefficients x and y in the Bezout Identity. The Extended Euclidean Algorithm is similar to Euclid's Algorithm, which we have talked about in the former part. The pseudo-code is shown below.

Algorithm 3: Extended Euclidean Algorithm

Input : Two integers a and b
Output: Their Bezout coefficients x and y and their GCD d

```
1  s ← 0
2  old_s ← 1
3  t ← 1
4  old_t ← 0
5  r ← b
6  old_r ← a
7  while r ≠ 0 do
8  |  quotient ← old_r div r
9  |  (old_r, r) ← (r, old_r − quotient · r)
10 |  (old_s, s) ← (s, old_s − quotient · s)
11 |  (old_t, t) ← (t, old_t − quotient · t)
12 end while
13 return (old_s, old_t) as Bezout coefficients. old_r as GCD.
```

The time complexity is the same as the Euclid's Algorithm, so it will not be discussed here.

Other Applications:

1. Finding multiplicative inverses in modular structures:

RSA is now the most widely used method for encryption. During the process of RSA, we need to calculate the private key d with the function $ed \equiv 1 (\text{mod } \varphi(n))$. We call this process calculating the multiplicative inverses in modular structures. The pseudo-code based on the Extended Euclidean Algorithm is shown below.

Algorithm 4: Calculating Multiplicative Inverse

Input : Two integers a and n

Output: The integer t that $at \equiv 1 (\text{mod } n)$

```
t ← 0
new_t ← 1
r ← n
new_r ← a
while new_r ≠ 0 do
    quotient ← r div new_r
    (t, new_t) ← (new_t, t − quotient · new_t)
    (r, new_r) ← (new_r, r − quotient · new_r)
end while
if r > 1 then
    return "a is not invertible"
end if
if t < 0 then
    t ← t + n
end if
return t
```

From the paragraph, we can see that if some a are not invertible, then the system can not work properly. To prevent this side effect, one must choose the a as a prime when using the RSA, otherwise there will be some number n that we can not get the private key.

2. Case for more integers:

We defined the GCD for more integers than 2 by the Bezout Identity: Suppose the sequence of numbers are $\{a_1, a_2, \cdots a_n\}$

- d is the smallest positive element satisfies there exists some $x_1, \cdots, x_n$ that $d = a_1x_1 + \cdots + a_nx_n$.
- Every number that satisfies this form should be the multiple of d.

We can also simplify this problem with some recursively defined relations like

$$gcd(a_1, a_2, \cdots, a_n) = gcd(a_1, gcd(a_2, \cdots, a_n))$$

3. Case for polynomials:

Bezout's Identity can also be used in the field of calculating the GCD of polynomials. We define a, b and g are polynomials that $au + by = g$, where u and v are polynomials and $deg(u) < deg(b) - deg(g)$ and $deg(v) < deg(a) - deg(g)$. The way of finding this GCD and the Bezout coefficients u and v are similar to the one of Extended Euclidean Algorithm, and just changing the quotient to some other such kind of calculation which saves the Euclidean Division instead of reminder only.

References

- Saugata Basu; Richard Pollack; Marie-Franoise Roy (2006). Algorithms in real algebraic geometry, chapter 4.2. Springer-Verlag.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Pages 859∼861 of section 31.2: Greatest common divisor.
- Knuth, Donald. The Art of Computer Programming. Addison-Wesley. Volume 2, Chapter 4.
- Rosen, Kenneth H.. Discrete Mathematics and Its Applications. Seventh Edition. Page 257 $\sim$ 274.