Chapter 00_04

Introducing Foundations

Elementary Number-Theoretic Notions

An application of number-theoretic algorithms is in ***cryptography***

- the discipline concerned with encrypting a message sent from one party to another, such that someone who intercepts the message will not be able to decode it.

Let the set Z = { …., -2, -1, 0, 1, 2, 3, ….} of integers.

Let the set N = {0, 1, 2, 3, ….} of natural numbers (nonnegative integers.

The notation $d \mid a$ (read "d ***divides*** a") means

- that $a = k*d$ for some integer k, (i.e., a is k multiple of d).

Outlines

- Relationship between GCD and *Division Theorem (Theorem 0.1)* {6, 7}
- Foundation to Euclid Algorithm for computing GCD {8-10}
- Estimate the time efficiency for Euclid Algorithm {12, 13}
- The use of Fibonacci numbers F_k, for analyzing Euclid Algorithm {14-16}
- Consecutive integer checking algorithm for computing gcd(x, y) {17-22}
- Prime Factorization in Middle-school procedure for computing gcd(x, y) {25-27}
- Use the Sieve of Eratosthenes to find all the primes which are less than a given number n and their time efficiency. [28-33]
- Several characteristics of Algorithms [34-35]
- Another Way for Finding the GCD: Writing a GCD as a Linear Combination [39-48]

We define $\gcd(0, 0) = 0$. This definition is necessary to make standard properties of the gcd function (such as $\gcd(x, 0) = |x|$) universally valid.

The following are elementary properties of the gcd function:

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

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

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

$\gcd(a, 0) = |a|$, and $\gcd(-5, 2) = \gcd(2, -1) = \gcd(-1, 0) = |-1| = 1 = \gcd(1, 0) = \gcd(2, 1) = \gcd(5, 1)$

$\gcd(a, ka) = |a|$, for $k \varepsilon Z = \{ \ldots., -2, -1, 0, 1, 2, \ldots\}$ of integers.

Lemma 0.1.1

If x is a positive integer, then gcd(x, 0) = x.

Proof:

Suppose x is a positive integer ($x \geq 0$). Certainly x is a common divisor of both x and 0 because x divides itself (i.e., $x \mid x$) and also x divides 0 (i.e., $x \mid 0$). Also no integer greater than x can be a common divisor of x and 0, (since no integer greater than x can divide x). Hence x is the greatest common divisor of x and 0.

QED

Relationship between GCD and *Division Theorem (Theorem 0.1)*

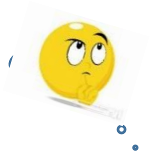
Lemma 0.1.2

If x and y are any integers not both zero, and if q and r are any integers such that

$$x = q * y + r, \ \ 0 \leq r < y,$$

then GCD(x, y) = GCD(y, r).

{i.e., ***GCD(x, y) = GCD(y, x mod y)***, since r = x mod y.}



Proof:
[The proof is divided into two parts: (1) proof that gcd(a, b) ≤ gcd(b, r), and (2) proof that gcd(b, r) ≤ gcd(a, b). Since each gcd is less than or equal to the other, the two must be equal.]

Example 0.30:

According to the previous Lemma 0.1.2, …

- If x = q * y + r (equivalently, x = q * y + x mod y), then GCD(x, y) = GCD(y, r).

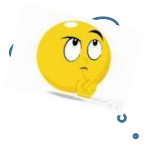
- Given x = 64, y = 24,

$$\underline{\mathrm{gcd}(x,\ y) = \mathrm{gcd}(y,\ r) \leftarrow x = q * y + (x \bmod y)}$$

$$\begin{aligned}
\mathrm{gcd}(64, 24) &= \mathrm{gcd}(24, 16) && \text{for } \underline{64} = 2 * \underline{24} + 16\\
&= \mathrm{gcd}(16, 8) && \text{for } \underline{24} = 1 * \underline{16} + 8\\
&= \mathrm{gcd}(8, 0) && \text{for } \underline{16} = 2 * \underline{8} + 0\\
&= 8 && \text{for } \underline{8} = 0 * \underline{0} + 8.
\end{aligned}$$

- This could express as GCD(x, y) = GCD(y, x mod y)
  = GCD(x mod y, y mod (x mod y))
  = GCD(y mod (x mod y), (x mod y) mod (y mod (x mod y)))
  …
- Note that gcd(24, 64) = gcd(64, 24)

Foundation to Euclid Algorithm for computing GCD



Given two integers x and y with ***$x \geq 0, y > 0$***

the Euclidean Algorithm computes GCD(x, y) based on two facts:

1. GCD(x, y) = GCD(y, x mod y),

   where $x = q * y + r$, $0 \leq r < y$,

   and $r = x \bmod y$.

2. GCD(x, 0) = x.

```
Algorithm Euclid(x, y)
//Compute gcd(x, y) by Euclidean algorithm
Input:    two non-negative x and y, not both zero integers
Output:  the greatest common divisor of x and y
   while (y ≠ 0) do {        |     if (y == 0)
       r ← x mod y;          |      then return x;
       x ← y;                |      else Euclid(y, x mod y);
       y ← r;}               |
   return x;                 |
```

GCD(x, y) = GCD(y, x mod y)

= GCD(x mod y, y mod (x mod y))

Several characteristics of Algorithms:

- …
- *[Different ways for specifying an algorithm]* The same algorithm can be represented in several different ways.
    - Euclidean algorithm can be defined recursively or non-recursively.
    - The nth Fibonacci Term can be computed recursively and iteratively.

Time efficiency for Euclid Algorithm

Example 0.32: Find the greatest common divisor of two integers, 7,276,500 and 3,185,325.

gcd(*7,276,500*, 3,185,325)

= gcd(3,185,325, *905,850*) … x mod y < x/2

= gcd(*905,850*, 467,775) … y mod (x mod y) < y/2

= gcd(467,775, *438,075*) … $< x/2^2$

= gcd(*438,075*, 29,700) … $< y/2^2$

= gcd(29,700, *22,275*) … $< x/2^3$

= gcd(*22,275*, 7,425) … $< y/2^3$

= gcd(7,425, *0*) … $< x/2^4$

= 7,425 …

Let y = 3,185,325 ≅ 2^{21}
$\log_2 y = \log_2 2^{21} = 21$
For the worse case, the number of recursive calls is about 21 times.

It requires 7 recursive calls to get the solution 7,425.

GCD(x, y) = GCD(y, x mod y)

```
Algorithm Euclid(x, y)
{ if (y = = 0)
  then return x;
  else Euclid(y, x mod y);}
```

Rationalize: Euclid(y, x mod y), where x mod y $\leq$ x/2.

- Let, say, x mod y = x/2.
- Euclid(y, x mod y) reduces x by one bit through right shift.
- Euclid(x mod y, y mod (x mod y)) where y mod (x mod y) $\leq$ y/2.
  - This reduces y by one bit through right shift.

GCD(x, y) = GCD(y, x mod y)
= GCD(x mod y, y mod (x mod y))

- This means, after two consecutive rounds, both arguments, x and y, are at the very least halved in value. (one right shift for each x and y.)
- Assume that both x and y are of *n*-bit integers.
  - The *base case* will be reached with *2n recursive calls*. i.e.,

gcd(x, y) = gcd(y, x mod y)
= gcd(x mod y, y mod (x mod y)).

G(109, 55) = G(55, 54)
109%55 = 54 $\leq \frac{109}{2}$
G(55, 54) = G(54, 1)
55%54 = 1 $\leq \frac{55}{2}$

- Then 2*n* recursive calls are needed. Each call involves quadratic-time O(n^2) division (Figure 1.2 in Ch 00_02).
- Therefore, the total running time is 2*n* * O(n^2) = O(n^3) .

The use of Fibonacci numbers F_k, for analyzing Euclid Algorithm

```
Algorithm Euclid(x, y)
{ if (y = = 0)
   then return x;
   else Euclid(y, x mod y);}
```

Observation:

- The overall running time of Euclid is proportional to the number of recursive calls it makes. (…2n recursive calls but not $\frac{x}{y}$.)

  Our analysis makes use of the Fibonacci numbers F_k, defined by the following recurrence:

$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2,$$

$$F_0 = 0, \quad F_1 = 1.$$

Lemma 0.1: If $x > y \geq 1$ and the invocation Euclid(x, y) performs $k \geq 1$ recursive calls, then

$$x \geq F_{k+2} \text{ and } y \geq F_{k+1},$$

where F_k is the kth number in the Fibonacci sequence.

```
Algorithm Euclid(x, y)
{if (y = = 0)
 then return x;
else Euclid(y, x mod y);}
```

Example:	F_0	F_1	...					F_7	F_8	...			
Fibonacci sequence is:	0	1	1	2	3	5	8	13	21	34	55	89	...
where k is:	0	1	2	3	4	5	6	7	8	9	10	11	...

For k = 5, $y \geq F_{k+1} = F_6 = 8$ and $x \geq F_{k+2} = F_7 = 13$. Select y be 8 and x be 13

GCD(13, 8) = GCD(8, 5) = GCD(5, 3) = GCD(3, 2) = GCD(2, 1) = GCD(1, 0) = 1.

Euclid(13, 649) = Euclid(649, 13%649) = Euclid(13, 649%13) = Euclid(12, 13%12) = Euclid(1, 12%1) = Euclid(1, 0) = 1.

Both requires 5 recursive calls.

The following theorem is an immediate corollary of Lemma 0.1.

Theorem 0.5 (Lame's Theorem):

For any integer $k \geq 1$, if $x > y \geq 1$ and $y < F_{k+1}$ then the call Euclid(x, y) makes fewer than k recursive calls.

Example:

$$F_k = F_{k-1} + F_{k-2} \quad k \geq 2 \text{ and } F_0 = 0 \text{ and } F_1 = 1.$$

Fibonacci sequence is:	0	1	1	2	3	5	8	13	21	34	55	89	…
where k is:	0	1	2	3	4	5	6	7	8	9	10	11	…

Let $y = 13$. $13 < F_{7+1}$ where $k = 7$

$E(21, 13) = E(13, 21\text{mod}13 = 8) = E(8, 13\text{mod}8 = 5) = E(5, 8\text{mod}5 = 3)$
$= E(3, 5\text{mod}3 = 2) = E(2, 3\text{mod}2 = 1) = E(1, 2\text{mod}1 = 0) = 0.$

It needs 6 recursive calls, fewer than $k = 7$ where $13 < F_{7+1} = 21$.

$E(649, 13)$ needs $5 < 7$ recursive calls to get $E(649, 13) = 1$

Consecutive integer checking algorithm for computing gcd(x, y)

Principle:

Let $t = \min\{|x|, |y|\}$ to be the possible common divisor for the given numbers x and y.

```
{
    { Test whether $t \mid x$.
      If not, reduce t by one, then test the new $t = t - 1 \mid x$.
    } Repeat this process until the new $t \mid x$. (The new $t = t - 1$).
    Test whether the new $t \mid y$.
    If yes, then t is the common divisor of x and y. [Found]
    Otherwise, reduce t by one.
} repeat the same process for the smaller t (at least one less than the current t) as the possible common divisor for the given number x and y.
```

Consecutive integer checking algorithm for computing gcd(x, y)

t = min{5, 0}; r = 5 mod 0 implies that

r = 5 – 0 – 0 - … - 0 operates infinitely.

```
Step 1:  t ← min{|x|, |y|}.
Step 2:  remainder = x mod t.
         If the remainder is 0, go to Step 3;
         Otherwise,  go to Step 4.
Step 3:  remainder = y mod t.
         If the remainder of y mod t is 0, return t as the answer and stop;
         otherwise, proceed to Step 4.
Step 4:  Decrease the value of t by 1. Go to Step 2.
```

Example 0.33: Find gcd(11, 3).

Let x = 11 and y = 3.

Step 1: t = min{ 11, 3} = 3.

Step 2: r = 11 mod 3 = 2 ≠ 0.
Go to Step 4

Step 4: t = t -1; that is, t = 3 – 1 = 2.
Go to Step 2.

Step 2: r = 11 mod 2 = 1 ≠ 0.
Go to Step 4.

Step 4: t = t -1; that is, t = 2 – 1 = 1.
Go to Step 2.

Step 2: r = 11 mod 1 = 0.
Then go to step 3

Step 3: r = y mod t; that is r = 3 mod 1 = 0;
then return t = 1 as the answer and stop.

gcd(-11, 3) =
gcd(|11|, 3) =
gcd(11, 3)

Let x = 11, y = 3.
t = min{11, 3} = 3
11%3 ≠ 0
11%2 ≠ 0
11%1 = 0
3%1 = 0 return 1
gcd(11, 3) = 1

X = 60, y = 24. Let t = 24.
60%24 ≠ 0
60%23 ≠ 0
60%22 ≠ 0
60%21 ≠ 0
60%20 = 0

24%20 ≠ 0

60%19 ≠ 0
60%18 ≠ 0
60%17 ≠ 0
60%16 ≠ 0
60%15 = 0

24%15 ≠ 0

60%14 ≠ 0
60%13 ≠ 0
60%12 = 0

24%12= 0
return t = 12
gcd(60, 24) = 12

- This algorithm does *not work correctly* when one of the inputs numbers is zero

  (why? On Step 2: $r = (3, 0) = 3$, then go to Step 4. $t = 0 – 1 = -1$. This fails to work!).

  This example illustrates why it is so *important to specify the range of an algorithm's input explicitly and carefully.*

- Definition: An algorithm is said to be *correct* if, for every input instance, it halts with the correct output.

Several characteristics of Algorithms:

- …
- *[Well-specified inputs' range]* The range of inputs for which an algorithm works *has to be specified carefully.*
    - Consecutive integer checking algorithm for computing gcd(x, y) does not work correctly when one of the input numbers is zero.

Consecutive_integer_checking_GCD(x, y)

Input: integers x and y, where one of the x and y ≠ 0

Output: GCD(x, y) = t.

```
public static int GCDCompute(int x, int y) {
        int t;
        x = Math.abs(x);
        y = Math.abs(y);
        if (x == 0) { return y;}      //if x = 0 and y = 0
        else if (y == 0) return x;    //and is not in the Cons_i_ck alg.

        if (x > y) {t = y;} else {t = x;}  //set t = min{x, y}

        while (!(x % t == 0)  ||  !(y % t == 0)){
                t = t - 1;}
        return t;
}// endGCDCompute()
```

Consecutive_integer_checking_GCD(x, y) //Is this correct?

Input: integers x and y, where one of the x and y ≠ 0

Output: GCD(x, y) = t.

```
public static int GCDCompute(int x, int y) {
        int t;
        x = Math.abs(x);
        y = Math.abs(y);
        if (x == 0) {return y;} else if (y == 0) {return x;}
        //t := min(|x|, |y|);
        if (x > y) {t = y;} else {t = x;}

        while (x % t != 0)   //what if x is not greater than y? don't worry
                {t = t - 1;}          //step 2.
        while (y % t != 0)
                {   t = t - 1;          //step 3.
                    while (x % t != 0)
                          {t = t - 1; }   //step 2.
                }
        return t;     //GCD(x, y) = t
}// endGCDCompute()
```

What is its time efficiency? t*O(n^2), where each mod takes O(n^2), and at most t number of mod to be executed.

NOT_Consecutive_integer_checking_GCD(x, y) //Is this correct?

Input: integers x and y, where one of the x and y ≠ 0

Output: GCD(x, y).

```
public static int GCDCompute(int x, int y) {
        int t;
        x = Math.abs(x);
        y = Math.abs(y);
        if (x == 0) {return y;} else if (y == 0) {return x;}

        //t := min(|x|, |y|);
        if (x > y) {t = y;} else {t = x;}

        while ( !((x % t == 0) && (y % t == 0)) ) {   //not ||
                while (!(x % t == 0))
                        {t = t - 1; }   //step 2
                if (!(y % t == 0)) {t = t - 1; }  //step 3
        }
        return t;
}// endGCDCompute()
```

What is its time efficiency?

Middle-school procedure for computing gcd(x, y)

Step 1 Find the prime factors of x (Use Sieve of Eratosthenes).

Step 2 Find the prime factors of y (Use Sieve of Eratosthenes).

Step 3 Identify all the common factors in the two primes expansion found in Step 1 and Step 2.

(If p is a common factor occurring p_x and p_y times in x and y, respectively, it should be repeated $\min\{p_x, p_y\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the given numbers.

Use assignment, conditional, loop, recursive and return statements to write an algorithm for this procedure.

Middle-school procedure for computing gcd(x, y)

Example 0.36: For the numbers 60, and 24, we get

x = 60 = 2.2.3.5 $\Longrightarrow$

y = 24 = 2.2.2.3

GCD(60, 24) = 2.2.3 = 12.

Step 3:

If $p = 2$, $p_x = 2$; $p_y = 3$; $\min\{p_x, p_y\} = \min\{2, 3\} = 2$;

If $p = 3$, $p_x = 1$; $p_y = 1$; $\min\{p_x, p_y\} = \min\{1, 1\} = 1$;

If $p = 5$, $p_x = 1$; $p_y = 0$; $\min\{p_x, p_y\} = \min\{1, 0\} = 0$;

Step 4: gcd(60, 24) is $2 * 2 * 3 * 1 = 12$

- This procedure is much more complex and slower than Euclid’s algorithm (inferior efficiency).
- The middle-school procedure does not qualify as a legitimate algorithm, because the prime factorization steps are not defined unambiguously.

Several characteristics of Algorithms:

- *[non-ambiguity] The non-ambiguity requirement* for each step of an algorithm cannot be compromised.
  - Prime Factorization in Middle School Procedure for computing gcd(x, y) is defined *ambiguously*
- …

Sieve of Eratosthenes (Ancient Greece, 200B.C.)

Problem: An algorithm for generating consecutive primes not exceeding any given integer n.

Step 1: Initialize a list of prime candidates with consecutive integers from 2 to n. {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, …, n}

Step 2: Continue the step of eliminating from the list all multiples of 2; then move on to the next item on the list, which is 3, and eliminate its multiples; then 5, until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed. {2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, …, n}

Example 0.37: Let n = 31. [p = 2, 3, 5]

if p =	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	2	3	**d**	5	**d**	7	**d**	9	**d**	11	**d**	13	**d**	15	**d**	17	**d**	19	**d**	21	**d**	23	**d**	25	**d**	27	**d**	29	**d**	31
3	2	3		5		7		**d**		11	r	13		**d**		17	r	19		**d**		23	r	25		**d**		29	r	31
5	2	3		5		7				11		13				17		19				23		**d**				29	r	31

- How far do you have to go for eliminating the non-prime numbers, for a given n?
- What is the largest number p whose multiples can still remain on the list?
    - e.g., for this case, the largest number p is 5 since $p = 5 \leq \sqrt{31}$.

Analysis of the problem:

- The following observation helps to avoid eliminating the same number more than once:
  - At the current pass, if p is a number, eliminate its multiples:
    - Begin to consider with the first multiple p*p, and then (p+1)*p, (p+2)*p, (p+3)*p, … where $p \le \lfloor\sqrt{n}\rfloor$.
    - The reason is that all its smaller multiples 2p, 3p, …, (p-1)p were eliminated on earlier passes through the list when the prime $p \le \lfloor\sqrt{n}\rfloor$, $p \in \{2, 3, 5, …, \lfloor\sqrt{n}\rfloor\}$. For example, when the current pass, if p = 5, both 2*p = 10 and 3*p = 15 were deleted in the earlier passes; namely, if p = 2, 5*2 = 10 was eliminated, and if p = 3, 5*3 = 15 was eliminated. Thus, for the pass, if p = 5, begin only with p*p, which is 5*5 = 25 ≤ n.
    - For example, for n = 48, consider p from 2, 3, …, up to the largest $p = 5 \le \lfloor\sqrt{48}\rfloor$. Then, when considering 5, only 5*5, ~~6*5~~, 7*5, ~~8*5~~, ~~9*5~~, are candidates to be eliminated. But 2*5, 4*5, 6*5, and 8*5 were eliminated when considering 2; and 3*5, and 9*5 were eliminated when considering 3. Thus, 30, 40, and 45 are not in the list when considering pass p = 5.

	25
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2
11	2
12	2
3	3
4	3
5	3
6	3
7	3
8	3
5	5

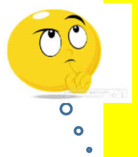
	51
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
…	2
25	2
3	3
4	3
5	3
6	3
---	3
17	3
5	5
6	5
7	5
8	5
9	5
10	5
7	7

Algorithm Sieve(n)

```
//Implements the sieve of Eratosthenes
Input: An integer n ≥ 2
Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n  do  A[p] ← p;   //Generate a copy of number from 2 on.
for p ← 2 to ⌊√n⌋  do {
          if A[p] ≠ 0  //p has not been eliminated on previous passes
          {          j ← p * p;
                     while j ≤ n do {
                               A[j] ← 0;  //mark the element as eliminated
                               j ← j + p; //j = p*p + p = (p+1)p}//end while_do
          } //end if (else back to for after p = p+1)
          p = p + 1;
         } //end for
//copy the remaining elements of A to array L of the primes
i ← 0;
for p ← 2  to n  do {
           if  A[p] ≠ 0  {
                     L[i] ← A[p];
                     i ← i + 1; } //end if
} //end for
return L
```

"if A[p] ≠ 0" checks whether p is a prime. If p is not a prime, then p*p, (p+1)*p, (p+2)*p, … ≤ n, then A[p*p], A[(p+1)*p], … have been eliminated on previous passes.

A[2] = 2, A[4] = 0 A[6]=0, A[8]=0, A[10]=0, A[12] =0, A[14]=0, A[16]=0, A[18]=0, A[20]=0, A[22]=0, A[24]=0, A[26]=0, A[28]=0, A[30]=0, ..., A[48]=0.
A[3]= 3, A[9]=0, A[12] = 0, A[15] = 0, A[18]=0, A[21] = 0, A[24] = 0, A[27]=0, A[30]=0, A[33]=0, …, A[45]=0, A[48]=0.
A[5] =5, A[25] = 0, A[30]=0, A[35]=0, A[40]=0, A[45]=0.



How fast the algorithm is?

- For n = 48, p*p ≤ 48 implies that p = 5, eliminate 23 of 2, 7 of 3, and 2 of 5 with total 32 numbers out of 48; for n = 49 implies that p = 7, eliminate 23 of 2, 7 of 2, 2 of 5 and 1 of 7 with total 33 numbers out of 49; for n = 63 implies that p = 7, eliminate 30 of 2, 10 of 3, 3 of 5 and 1 of 7 with total 44 numbers out of 63;
- The total number **d** eliminated out of ⌊p*p⌋ ≤ n ≤ ⌊(p+1)*(p+1)⌋ is *at least*

$$\mathbf{(1_2 + 2_3 + 3_4 + 4_5 + 5_6 + 6_7 + 7_8 + \ldots + (p-1)_p\ )} = \frac{(\boldsymbol{p}-\mathbf{1})(\boldsymbol{p}-\mathbf{1}+\mathbf{1})}{\mathbf{2}}$$

$$= \frac{\boldsymbol{p}(\boldsymbol{p}-\mathbf{1})}{\mathbf{2}} \le \mathbf{d} \le \sum_{\boldsymbol{p'}=\mathbf{2}}^{\lfloor\sqrt{\boldsymbol{n}}\rfloor}\left(\left\lfloor\frac{n}{p'}\right\rfloor - (p'-1)\right).$$

2*2, 3*2, 4*2, 5*2, 6*2, 7*2, 8*2, …, p*2 ≤ n

3*3, 4*3, 5*3, 6*3, 7*3, 8*3, …, p*3 ≤ n

4*4, 5*4, 6*4, 7*4, 8*4, …, p*4 ≤ n

5*5, 6*5, 7*5, 8*5, …, p*5

6*6, 7*6, 8*6, …, p*6

7*7, 8*7, …, p*7

8*8, …, p*8

if p = 7, there are p-1 in this column (which have been eliminated)

$$1 + 2 + 3 + 4 + 5 + 6 = \frac{\mathbf{7(7-1)}}{\mathbf{2}} = 21$$

Time Efficiency for the Sieve of Eratosthenes to find all the prime which is less than n.

The total multiplications required is $\frac{\boldsymbol{p(p-1)}}{\boldsymbol{2}} \le \mathbf{m} \le \sum_{\boldsymbol{p'=2}}^{\lfloor\sqrt{\boldsymbol{n}}\rfloor}(\lfloor\frac{n}{p'}\rfloor - (p'-1)) < \mathrm{n}$

where $p' \in \{2, 3, 5, 7, \dots, \lfloor\sqrt{n}\rfloor\}$.

If the running time for each multiplication is $O(n^2)$,
the running time for the algorithm is $n * O(n^2) = O(n^3)$.

What is $\sum_{\boldsymbol{p'=2}}^{\lfloor\sqrt{\boldsymbol{n}}\rfloor}(\lfloor\frac{n}{p'}\rfloor + (p'+1))$ when $n = 63$?

Total number of eliminations is $\sum_{\boldsymbol{p'=2}}^{\lfloor\sqrt{\boldsymbol{n}}\rfloor}(\lfloor\frac{n}{p'}\rfloor - (p'-1))$

$= (\lfloor\frac{63}{2}\rfloor - (2-1)) + (\lfloor\frac{63}{3}\rfloor - (3-1)) + (\lfloor\frac{63}{5}\rfloor - (5-1)) + (\lfloor\frac{63}{7}\rfloor - (7-1))$

$= 30_2 + 19_3 + 8_5 + 3_7 = 60$ times marking "A[j] ← 0;" in Algorithm Sieve(n), which generates the prime numbers 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61.

What this says is that when considering 3, p*p = 3*3 is the first one A[9] ← 0 . Then the next is 4*3, A[12] ← 0, but this was done when considering 2. That is, when $(p+1)*3 \le 63$. A[(p+1)*3] ← 0 even when (p+1)*3 is even. Repetition of eliminations are occurred.

Several characteristics of Algorithms:

- [non-ambiguity] The non-ambiguity requirement for each step of an algorithm cannot be compromised.
  - Prime Factorization in Middle School Procedure for computing gcd(m, n) is defined ambiguously
- [Well-specified inputs' range] The range of inputs for which an algorithm works has to be specified carefully.
  - Consecutive integer checking algorithm for computing gcd(m, n) does not work correctly when one of the input numbers is zero.
- [Different ways for specifying an algorithm] The same algorithm can be represented in several different ways.
  - Euclid's algorithm can be defined recursively or non-recursively.
  - The nth Fibonacci Term can be computed recursively and iteratively.

Several characteristics of Algorithms:

- [Several algorithms for a problem] Several algorithms for solving the same problem may exist.
  - Euclid, Consecutive integer checking and Middle school procedure for computing gcd(m, n).
- [Various Speeds of different Algorithms for solving the same problem] Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.
  - An exponential *Algorithm Fibonacci_Number_F(n)* computes recursively the list of the n Fibonacci members based on its definition, and
  - a polynomial *Polynomial_Algorithm_Fib(n)* computes non-recursively the list of its of its n members.

H

Another Way for Finding the GCD:
Writing a GCD as a Linear Combination

- Definition:

  Any integer d is said to be a *linear combination of integers* x and y if, and only if there exist integers i and j such that i*x + j*y = d.

- Write a GCD as a Linear Combination

Theorem 0.2

If d | a and d | b, then for integers i and j, d | (ia + jb).

Example:

Consider a = 24 and b = 30.

Since 6 | 24 and 6 | 30, then for integers i and j, 6 | (i*24+j*30).

That is, (i*24+j*30) is multiples of 6.

The fact is that 24 = 4*6 and 30 = 5*6;

(i*24+j*30) = i*4*6 + j*5*6 = (i*4 + j*5)6.

That is, 6 | (i*24+j*30) for 6 | (i*4 + j*5)6.

(i*24+j*30) has *the least multiple of 6 when (i*4 + j*5) = 1.*

- Definition:

  Any integer d is said to be a *linear combination of integers* x and y if, and only if there exist integers i and j such that i*x + j*y = d.

- Write a GCD as a Linear Combination

Theorem 0.1.4.5: (Writing a GCD as a Linear Combination)

For all integers a and b, not both zero, if d = gcd(a, b), then there exist integers i and j such that a*i + b*j = d.

Theorem 0.1.4.5: (Writing a GCD as a Linear Combination)

For all integers a and b, not both zero, if d = gcd(a, b),
then there exist integers i and j such that a*i + b*j = d.

Example: (Write a GCD as a Linear Combination. Is this a way to find the gcd(30, 24)?)

Consider a = 30 and b = 24. $\underline{\text{x = q * y + r yields x – q * y = r}}$

Then d = gcd(30, 24) = gcd(24, 6) ↔ $\underline{30}$ = 1 * $\underline{24}$ + 6 yields (1*$\underline{30}$ + -1*$\underline{24}$) = 6. (1)

= gcd(6, 0) ↔ $\underline{24}$ = 4 * $\underline{6}$ + 0 yields (1*$\underline{24}$ + -4 * $\underline{6}$) = 0. (2)

= 6. ↔ $\underline{6}$ = 1 * $\underline{0}$ + 6 yields (1* $\underline{6}$ + -1 * $\underline{0}$) = 6. (3)

6 $=_{(3)}$ (1* $\underline{6}$ + -1 * $\underline{0}$)

$=_{(2)}$ (1* $\underline{6}$ + -1* (1 * $\underline{24}$ + -4 * $\underline{6}$)) = (-1 * $\underline{24}$ + 5 * $\underline{6}$)

$=_{(1)}$ (-1 * $\underline{24}$ + 5* (1*$\underline{30}$ + -1*$\underline{24}$)) = (-1 * $\underline{24}$ + 5 * $\underline{30}$ + -5 * $\underline{24}$))

= (5 * $\underline{30}$ + -6 * $\underline{24}$)

= 6 (5 * 5 – 6 * 4), where (5 * 5 – 6 * 4) = 1, i = 5 and j = -6.

Thus, gcd(24, 30) = 6 → there exist 5 and -6 integers, such that 6 (5 * 5 – 6 * 4) = 6.

Example 0.26a: Writing a GCD as a Linear Combination.

Use the Euclid algorithm to find the gcd of 30 and 24, which is 6.

gcd(30, 24) = gcd(24, 30 mod 24) = gcd(24, 6) = gcd(6, 0) = 6.

Express the computation of gcd(30, 24) as a linear combination of 30 and 24:

- The first three steps use successive applications of the quotient-remainder theorem. Then, find the coefficients of the linear combination by substituting back through the results of the previous three steps.

gcd(30, 24) = gcd(24, 6) ↔ $\underline{30} = 1 * \underline{24} + 6$, yields $6 = \underline{30} - 1 * \underline{24}$. (1)

gcd(24, 6) = gcd(6, 0) ↔ $\underline{24} = 4 * \underline{6} + 0$, yields $0 = \underline{24} - 4 * \underline{6}$. (2)

gcd(6, 0) = 6 ↔ $\underline{6} = 1 * \underline{0} + 6$, yields $6 = \underline{6} - 1 * \underline{0}$. (3)

Since $6 = \underline{6} - 1 * \underline{0} = 1 * \underline{6}$ from step (3)

$= 1 * (\underline{30} - 1 * \underline{24})$ by substitution of $\underline{6}$ from step (1)

$= 1 * \underline{30} + (-1) * \underline{24}$

Thus, gcd(30, 24) = 1*30 + (-1)*24

= 6(1*5 + (-1)*4) = 6, where (1*5 + (-1)*4) = 1, i = 1 and j = -1.

The following theorem states that if x and y are any integers, not both zero, then $\gcd(x, y)$ is the *smallest* positive element of the set $\{ix + jy \mid i, j \in Z, ix + jy > 0\}$ of linear combination of x and y.

Theorem 0.3

Let x and y be integers, not both 0.

Let $d = \min\{ix + jy \mid i, j \in Z \text{ and } ix + jy > 0\}$, which means that

d is the *smallest* positive linear combination of x and y.

Then, $d = \gcd(x, y)$.

Rationalize:

Let $d = \gcd(x, y)$. This says, $d \mid x$ and $d \mid y$ and $d \mid (x+y)$.

$d \mid (md + nd)$ where $x = md$ and $y = nd$.

$d \mid (im + jn)d$ and $imd + jnd > 0$.

Theorem 0.3

Let x and y be integers, not both 0. Let

$$d = \min\{ix + jy \mid i, j \in Z \text{ and } ix + jy > 0\}.$$

i.e., d is the smallest positive linear combination of x and y.

Then, d = gcd(x, y).

Example:

Let x = 30 and y = 24.

$d = \min\{-1*24+1*30,\ -2*24 + 2*30,\ 3*24 + (-2)*30,\ \ldots,\ 2*24 + (-1)*30,$

$\quad -3*24 + 3*30,\ 1*24 + 0*30,\ 0*24 + 1*30,\ \ldots,\ -6*24 + 5*30,\ \ldots$

$\quad \mid i, j \in Z \text{ and } i*24+j*30 > 0\}.$

$i * 30 + j * 24 > 0$ if and only if $(i * 5 + j * 4)6 > 0$.

For yielding the minimum value of d (i.e., the smallest positive linear combination of x = 30 and y = 24, $i * 30 + j * 24 > 0$), $(i * 5 + j * 4)6 > 0$ has a minimum value if and only if $i * 5 + j * 4 = 1$, where $i, j \in Z$.

This implies that i = 1 and j = -1 can be a choice.

Then d = gcd(30, 24) = 6.

Theorem 0.2

If $d \mid a$ and $d \mid b$, then for integers i and j, $d \mid (ia + jb)$

Corollary 0.3.1

For any integers x and y, if $d \mid x$ and $d \mid y$, then $d \mid \gcd(x, y)$.

This Corollary 0.3.1 states:

Suppose x and y are integers, not both 0. Then every common divisor of x and y is a divisor of $\gcd(x, y)$.

Greatest Common Divisor (GCD)

Describe Euclid's algorithm for efficiently computing the greatest common divisor of two given integers, not both zero.

Theorem 0.4 (GCD Recursion Theorem)

For any nonnegative integer a and any positive integer b,

gcd(a, b) = gcd(b, a mod b).

This theorem gives us a straightforward method for determining the greatest common divisor of two integers.

Euclid’s Algorithm for Computing Common Divisor

Euclid’s algorithm for computing gcd(m, n):

```
repeatedly applying the equality
    gcd(m, n) = gcd(n, m mod n),
    //where m mod n is the remainder of $\frac{m}{n}$,
until (m mod n) = 0.
```

Since gcd(m, 0) = m, the last value of m is the greatest common divisor of the initial m and n.

This method is called Euclid’s Algorithm, developed by Euclid around 300 B.C.

gcd(60, 24) d=12= min{1*60 + (-2)*24, 3*60 + (-7)*24, …. }

Note that if gcd(x, y) = d, then {ix + jy} = {d $(\frac{x}{d}\, i + \frac{y}{d} j\,)$}.

Example 0.28: find gcd(60, 24).

Let x = 60, y = 24. To find gcd(60, 24), write gcd(60, 24) as a linear combination of 60 and 24.

By Theorem 0.3, gcd(x, y) = d = min{ix + jy | i, j ∈ Z and ix + jy > 0}.

For i, j ∈ Z and ix + jy > 0 implies i * 60 + j * 24 = 12(i * 5 + j * 2) > 0.

This implies that (i * 5 + j * 2) > 0.

*12(i * 5 + j * 2) > 0 has a minimum value if* 0 < (i * 5 + j * 2) = 1.

That is, (i, j) ∈ {(1, -2), (-1, 3), (3, -7), (-3, 8), (5, -12), (-5, 13), (7, -17), (-7, 18), etc. | (i * 5 + j * 2) = 1, and (i*60 + j *24) > 0}.

gcd(60, 24) d=12= min{1*60 + (-2)*24, 3*60 + (-7)*24, …. }

Note that if gcd(x, y) = d, then {ix + jy} = $\{d\,(\frac{x}{d}\,i + \frac{y}{d}\,j\,)\}$.

Example 0.28: find gcd(60, 24). continue….

gcd(60, 24)

$= \gcd(24, 12)$ for $\underline{60} = 2 * \underline{24} + 12$ which yields $12 = 1* \underline{60} - 2* \underline{24}$ (1)

$= \gcd(12, 0)$ for $\underline{24} = 2 * \underline{12} + 0$ which yields $0 = 1* \underline{24} - 2* \underline{12}$ (2)

$= 12$ for $\underline{12} = 0 * \underline{0} + 12$ which yields $12 = 1* \underline{12} - 0* \underline{0}$ (3)

$12 = 1 * \underline{12} – 0 * \underline{0}$ from (3)

$= 1 * \underline{12} – 0 * (1 * \underline{24} – 2 * \underline{12}) = 1 * \underline{12}$ from (2)

$= 1 * (1 * \underline{60} – 2 * 24)$ from (10

$= 1 * \underline{60} – 2 * \underline{24} = 12\{1* 5 + (- 2)* 2 \} > 0.$

Thus, GCD(60, 24) = 1*60 + (-2)*24 = 12(1*5 + (- 2)*2) = 12.

Example 0.30:

note that gcd(24, 64) = gcd(64, 24)

a = q*b + (a mod b),
64 = 2 * 24 + 16
24 = 1*16 + 8
16 = 2 * 8 + 0
8 = 0 * 0 + 8

According to the theorem 0.4 (GCD Recursion Theorem),

gcd(64, 24)

= gcd(24, 16), where 16 = 64 mod 24.

for $\underline{64}$ = 2 * $\underline{24}$ + 16 yields 16 = 1*$\underline{64}$ – 2*$\underline{24}$ (1)

= gcd(16, 8) for $\underline{24}$ = 1 * $\underline{16}$ + 8 yields 8 = 1*$\underline{24}$ – 1*$\underline{16}$ (2)

= gcd(8, 0) for $\underline{16}$ = 2 * $\underline{8}$ + 0 yields 0 = 1*$\underline{16}$ – 2* $\underline{8}$ (3)

= 8. for $\underline{8}$ = 0 * $\underline{0}$ + 8 yields 8 = 1* $\underline{8}$ – 0* $\underline{0}$ (4)

To compute the linear combination, we begin

8 = 1 * $\underline{8}$ – 0 * 0 using (4)

= 1 * $\underline{8}$ – 0*(1*$\underline{16}$ - 2 * $\underline{8}$), using (3): 0 = 1*$\underline{16}$ – 2* $\underline{8}$ for 0

= 1 * (1*$\underline{24}$ – 1 *$\underline{16}$), using (2): 8 = 1*$\underline{24}$ – 1*$\underline{16}$ for 8

= 1*$\underline{24}$ – 1 *(1*$\underline{64}$ -2*$\underline{24}$) using (1): 16 = 1*$\underline{64}$ – 2*$\underline{24}$ for 16

= -1 * 64 + 3 * 24 which is the linear combination.

gcd(64, 24) d=8= min{(-1)1*64 + (3)*24, 2*64 + (-5)*24, …. }

Note that if gcd(x, y) = d, then {ix + jy} = {d $(\frac{x}{d}\ i + \frac{y}{d} j\ )$}.

Example 0.30: find gcd(64, 24).

Let x = 64, y = 24. To find gcd(64, 24), write gcd(64, 24) as a linear combination of 64 and 24.

By Theorem 0.3, gcd(x, y) = d = min{ix + jy | i, j ∈ Z and ix + jy > 0}.

For i, j ∈ Z and ix + jy > 0 implies i * 64 + j * 24 = 8(i * 8 + j * 3) > 0.

This implies that (i * 8 + j * 3) > 0.

*8(i * 8 + j * 3) > 0 has a minimum value if* 0 < (i * 8 + j * 3) = 1.

That is, (i, j) ∈ {(-1, 3), (2, -5), ~~(3, -7), (4, -11),~~ (5, -13), (8, -21), …|
(i * 8 + j * 3) = 1, and (i*64 + j *24) > 0}.

Prime Factorization and Relative Prime

Cryptography – The RSA Public Key Cryptosystem

The Rivest-Shamir-Adleman (RSA) cryptosystem uses all the ideas we have introduced in this lecture note. It derives very strong guarantees of security by ingeniously exploiting the wide gulf between the polynomial-time computability of certain number-theoretic tasks: (

- modular exponentiation,
- greatest common divisor,
- primality testing) and
- the intractability of others (factoring).