Οικονομικό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής
ΠΜΣ στα Πληροφοριακά Συστήματα

Κρυπτογραφία και Εφαρμογές

Μαριάς Ιωάννης
marias@aueb.gr

Μαρκάκης Ευάγγελος
markakis@gmail.com

Summary

- Primality Testing
  - ✓ Density of primes
  - ✓ Eratosthenes' sieve
  - ✓ Trial division
  - ✓ Fermat test
  - ✓ Miller-Rabin test
  - ✓ Other algorithms: Solovay-strassen, deterministic algorithms
- Integer Factorization
  - ✓ Pollard's rho method

Primality Testing

In public key cryptography we often need to solve the following problem:

- Pick a prime number p within a certain range, e.g. a prime with up to 512 bits

1. How many numbers do we need to try till we find a prime?
2. Given a number how do we test that it is a prime?

Primality Testing

- Density of primes
  - ✓ Prime numbers are not sparse.
  - ✓ **Chebyshev's theorem (1850):** there is always a prime between n and 2n
  - ✓ Density function π(n) = the number of primes between 2 and n
    - e.g. π(10) = 4 → 2,3,5,7
  - ✓ **Prime number theorem (1896):** The density function π(*n*) satisfies:

    $\lim_{n \to \infty} (\pi(n) / (n / \ln n)) = 1$, or esle $\pi(n) \approx n / \ln n$ *for large enough* n
  - ✓ Example
    - $n = 10^9$
    - $\pi(n) = 50{,}847{,}534$
    - $n / \ln n \approx 48{,}254{,}942$
    - Deviation 6%

- Density of primes
  - By the prime number theorem:
  - Prob(randomly chosen integer between 1 and n is prime) = 1/ ln *n*
  - Hence if we examine about ln *n* randomly chosen integers between 1 and n, one of them will be prime with high probability
  - To find a 512-bit prime we can check about $\ln 2^{512} \approx 355$ randomly chosen integers of 512-bits
  - **BUT:** once we choose a number, how do we really check that this is a prime number?

Primality Testing

- The sieve of Eratosthenes (3rd century B.C.)
  - ✓ A method to identify all primes up to a given number n
  - ✓ The algorithm:
  - ✓ **Input:** An integer n ≥ 2
  - ✓ **Output:** find all primes < n
  - ✓ Idea: Consider a boolean array a of size n representing if a number is prime or not
  - ✓ Initially all entries are true
    - Gradually non-primes will become false
  - ✓ Starting from number 2 and going up to n-1
    - If a[x]=false go to next element
    - Else x is prime and set all its multiples (that are < N) to false

- The sieve of Eratosthenes (3rd century B.C.)

```
for (int i = 2; i < N; i++)
     a[i] = true;
for (int i = 2; i < N; i++)
     if (a[i])
          for (int j = i; j*i < N; j++)
            a[i*j] = false;//multiples of i
                           //are not prime numbers
```

1. Why don't we do anything when a[i]= false?

   By Euclid's theorem, every number can be written as a product of prime numbers. It suffices to filter out only the multiples of prime numbers.

2. Why does the loop begin from i^2 ?

   If $x<i^2 = i*i$, and x is not a prime, then x has some prime factor <i-1. Hence a[x] became false in some previous iteration

Primality Testing

- From now on we focus on testing whether a particular number n is prime
- We may assume n is odd
- Trial division
  - ✓ Try to see if any of the numbers 2, 3, 4,…,n-1 divides n
  - ✓ Actually it suffices to try only with the numbers 2, 3, ..., $\lfloor\sqrt{n}\rfloor$
    - If n is composite it has a factor, which is at most $\sqrt{n}$
  - ✓ In fact, since n is odd, we can also remove the even numbers
  - ✓ Worst case complexity: $\sqrt{n}/2$, hence $O(\sqrt{n})$
  - ✓ Exponential since $\sqrt{n} = 2^{\log n/2}$
    - Effective only for small values of n
    - For RSA, n is 512 bits long or even longer

- Pseudo prime numbers
  - ✓ Recall Fermat's little theorem:
  - ✓ If n is prime then $a^{n-1} \equiv 1 \pmod{n}$ for every $a \in \{1,\dots,n-1\}$
  - ✓ For a given $a \in \{1,\dots,n-1\}$, a number n is a **base-a pseudoprime** if n is composite and :

$$a^{n-1} \equiv 1 \pmod{n} \quad (*)$$

  - ✓ Hence if we find a number a for which this does not hold, certainly n is composite
  - ✓ If we picked an a for which (*) holds , we *hope* n is prime, i.e., we hope there cannot be too many composites that can satisfy (*)

- **Fermat Test**

```
Algorithm PSEUDOPRIME(n) //n is an odd integer
Pick a positive integer 1≤a<n at random
if   a^(n-1)  ≡ 1 (mod n) then return PRIME  // pass test
                           else return COMPOSITE
```

- Computing a^{n-1} (modn) should be done with the algorithm for modular exponentiation
- One can run the algorithm for some fixed a, e.g., a=2
- The algorithm can make errors but only of one kind:
  - ✓ If it says that n is composite, then it is correct
  - ✓ If it says that n is prime then it is wrong only in the case that n is a base-a pseudoprime

Primality Testing

- How often is the algorithm wrong?
  - Rarely.
  - For a=2: there are only 22 values of *n* in [1, 10,000] for which the algorithm fails. The first 4 are 341, 561, 645, και 1105.
  - $341 = 11 * 31$ and $2^{340} \equiv 1 (\bmod 341)$

- Estimates for base-2 pseudoprimes
  - For a 512-bit randomly chosen number that the algorithm thinks it is prime, the probability that the number is a base-2 pseudoprime is roughly $1/10^{20}$
  - For a 1024-bit randomly chosen number that the algorithm thinks it is prime, the probability that the number is a base-2 pseudoprime is roughly $1/10^{41}$

- Carmichael numbers
  - Actually due to Korselt
  - They are the composite numbers that pass the test *for all* a's
  - Alternative definition: A number n is a Carmichael number if it is not divisible by the square of a prime (square-free) and for all prime divisors p of n, it is true that $p-1 \mid n-1$
  - They are extremely rare (561, 1105, 1729, 2465,…)
  - $561 = 3 \cdot 11 \cdot 17$
  - There are only 255 of them less than 10^8
  - There are 20,138,200 Carmichael numbers between 1 and 10^{21} (approximately one in 50 billion numbers)

Primality Testing

- Theorem: if a number n fails the Fermat test for some value of a then n also fails for at least half of the choices of $a<n$

- If we ignore Carmichael numbers for now then:
- Pr[PSEUDOPRIME(n) returns PRIME, when n is COMPOSITE] $\leq 1/2$

- If we repeat the algorithm k times by choosing k different values for a, say $\alpha_1, \alpha_2, \ldots, \alpha_k$, then
- Pr[PSEUDOPRIME(n) returns PRIME, when n is COMPOSITE] $\leq 1/2^k$

- Miller-Rabin randomized primality test
  - ✓ It modifies and improves PSEUDOPRIME(n)
  - ✓ It is also based on Fermat's little theorem
  - ✓ Definition: A number $x \in Z_n$ is a square root of y modn if $x^2 \equiv y$ modn
  - ✓ Lemma: If n is prime, the only square roots of 1 modn are +1, -1 modn
  - ✓ If n is an odd number, write n-1 in the form $n-1 = 2^k m$, for some k
  - ✓ Then by Fermat's theorem, if n is prime, $a^{(n-1)/2}$ is a square root of 1 modn (and hence it is either +1 or -1 modn)
  - ✓ The algorithm is based on the fact that if we keep taking square roots and n is prime,
    - Either we hit a -1 modn at some point
    - or we will keep seeing 1 modn till the end ($a^m = 1$ modn)

Primality Testing

- Miller-Rabin randomized primality test
  - MILLER-RABIN(n)

```
Suppose n-1 = 2^k m, where k ≥ 1 and m is odd
Choose a random integer a with 1≤a≤n-1
Compute b = a^m modn /*by the algorithm MODULAR-
EXPONENTIATION that we saw in previous lectures*/
if b ≡ 1 modn then return PRIME
for i=0 to k-1 do
    if b ≡ -1 modn return PRIME
    else
        b = b^2 modn
return COMPOSITE
```

Analysis

- **Part (a):** We first show that when the algorithm says COMPOSITE, it is correct
- Suppose for the sake of contradiction that n is a prime number and the program answers COMPOSITE
- Then for every i with 0≤ *i* ≤ k-1, we have that

$$a^{2^i m} \neq -1 \bmod n$$

- Since n is prime we also have that

$$a^{2^k m} = 1 \bmod n$$

- This means that $a^{2^{k-1} m}$ is a square root of 1 modn

Analysis

- By our assumptions it follows that

$$a^{2^{k-1}m} = 1 \bmod n$$

- But then $a^{2^{k-2}m}$ is also a square root of 1 modn
- Continuing by using the same argument we eventually conclude that a^m = 1 modn, a contradiction since then the algorithm would have answered PRIME
- **Part (b):** When the program answers PRIME, there is a chance that n is composite.
- It has been shown that the error chance is at most ¼
- Hence by choosing multiple random numbers a_1, a_2,…,a_s and repeating the process the error rate falls down to $1/4^s$

Example

- Let $n = 221$, $n-1= 2^2 \cdot 55$ (k=2, m=55)
- Let $a = 137$
- $a^{55} \bmod 221 = 188 \neq 1 \bmod 221$
- $a^{110} \bmod 221 = 205 \neq -1 \bmod 221$
- Hence the base a=137 is a witness for the compositeness of 221
- Note that a primality testing algorithm does not necessarily reveal the factors of a composite number!

Complexity

- The only non-trivial operations are raising to powers modn
- Hence if we use the algorithm of repeated squaring, running time is polynomial $(O(\log n)^3)$

Primality Testing

- Other randomized tests: [Solovay-Strassen ’77], Miller-Rabin perfoms better though
- If Generalized Rieman hypothesis is true, Miller-Rabin can be turned into a deterministic algorithm
- [Agrawal, Kayal, Saxena 2002]: The first deterministic polynomial time primality test (it was an open problem for many years)
- First analysis $O((\log n)^{12})$
- Later improved to $O((\log n)^{6})$
- Still impractical to use
- Randomized tests still better in practice

Integer Factorization

- One of the most important problems in Cryptography
- State of the art
  - ✓ May 2005: factorization of RSA-200 (663 bits, 200 decimal digits)
  - ✓ November 2005: factorization of RSA-640 (640 bits, 193 decimal digits), 5 months on 80 2.2GHz processors
  - ✓ Dec 2009: factorization of RSA-768 (768 bits, 232 decimal digits), took almost 2 years with hundreds of machines.
    Research team: Kleinjung, Aoki, Franke, Lenstra, Thome, Gaudry, Kruppa, Montgomery, Bos, Osvik, te Riele, Timofeev, Zimmerman
  - ✓ Up to now, 16 of the 54 challenge numbers have been factored
  - ✓ For updates on the RSA factoring challenge (not active any more by the RSA labs) see

    http://en.wikipedia.org/wiki/RSA_numbers

    http://www.rsa.com/rsalabs/node.asp?id=2092

Integer Factorization

- Statement of the problem:
- Given an odd integer n, find one non-trivial factor of n
  - ✓ We may assume that n is composite (e.g. by first running a primality test on n)
  - ✓ An efficient algorithm should be polynomial in logn

- The most interesting case for public key cryptography is when n = pq for primes p, q of around the same size (512 bits)
- Definition: A composite number of the form n = pq, where p, q are primes, is called semi-prime
- Up to now we do not know if there exists a polynomial time algorithm for the problem

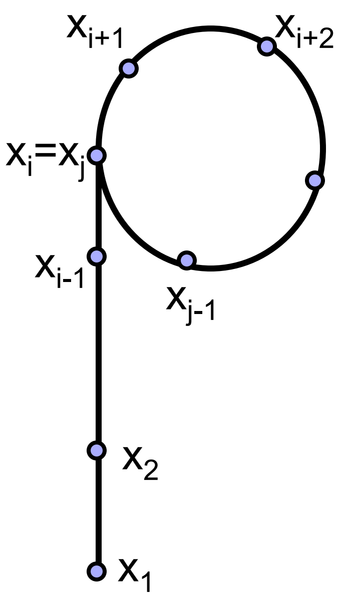
- Factoring algorithms
  - ✓ Most naive approach: trial division
    - Works in time $O(\sqrt{n})$
  - ✓ Many other approaches have been suggested
  - ✓ Here we will only see the rho-heuristic by Pollard (1975)
  - ✓ Let p be the smallest prime factor of n
  - ✓ Idea:
  - ✓ Suppose there exist $x_i, x_j \in Z_n$ such that $x_i \neq x_j$ but $x_i \equiv x_j \bmod p$
  - ✓ Then $\gcd(x_i - x_j, n)$ is a non-trivial factor
  - ✓ How can we find such x_i, x_j?

Integer Factorization

- ✓ We will try to choose a subset $X \subseteq Z_n$ and then compute $gcd(x_i-x_j,n)$ for every pair $x_i,x_j \in X$ (X should not be too large)
- ✓ POLLARD-RHO actually helps in reducing the number of required gcd computations
- ✓ Let $f(x)=x^2+\alpha$ (usually a = -1 or +1)
- ✓ Consider the transformation $x \rightarrow f(x) \bmod n$
- ✓ Suppose x_1 is a random element of Z_n and consider the sequence $X = \{x_1, x_2, x_3, x_4 \ldots\}$ defined by $x_j = f(x_{j-1}) \bmod n$
- ✓ Since we are in Z_n, this is a finite sequence, beyond some point it repeats itself, i.e., $\exists i,j$ such that $x_i \equiv x_j \bmod n$, $x_{i+1} \equiv x_{j+1} \bmod n$,…
- ✓ By birthday paradox X has about $\sqrt{n}$ elements if f is a random enough function

Integer Factorization

- Consider the graph G with vertices the values x_i modn and edges the consecutive pairs in the sequence
- The graph has a tail and a circle (forms a rho)
  - ✓ x_i modn -> x_{i+1} modn, -> … -> x_j modn ≡ x_i modn
- Basic idea of **POLLARD-RHO(n)** is to find a collision, i.e., a pair x_i , x_j such that $x_i \neq x_j$ but $x_i \equiv x_j$ modp



✓ Since we do not know p we may need to check all possible pairs, x_i, x_j

✓ We will end up checking pairs inside the cycle

✓ Hence we would need to check if

$$1 < \gcd(x_i - x_j, n) < n$$

- Pollard's heuristic
  - **POLLARD-RHO(n)**

```
i ← 1
x_1 ← RANDOM(0, n - 1)
y ← x_1
k ← 2
while TRUE do
      i ← i + 1
      x_i=(x^2_i-1 -1)mod n
      d ← gcd(y - x_i, n)
      if d ≠ 1 and d ≠ n
         then print d
      if i = k
         then y ← x_i//y takes only the values x_1, x_2, x_4, x_8 …
              k ← 2k
```

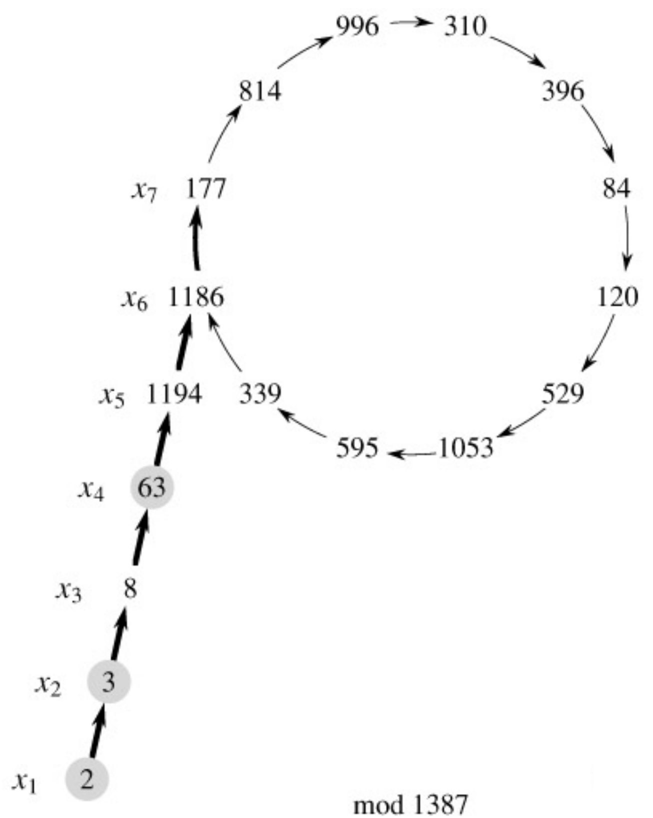
Analysis

- Note that the algorithm never prints a wrong answer
- But it may keep on going without ever printing something

- The variable y takes only the values x_1, x_2, x_4, x_8,…
- The gcd computations that we perform are
  - ✓ gcd(x_1 - x_2, n) (when y = x_1)
  - ✓ gcd(x_2 - x_3, n), gcd(x_2 - x_4, n) (when y = x_2)
  - ✓ gcd(x_4 - x_5, n), gcd(x_4 - x_6, n), gcd(x_4 - x_7, n), gcd(x_4 - x_8, n) (when y = x_4)
  - ✓ …
- If we wait long enough, y will enter the cycle
  - ✓ Birthday paradox cannot really be formally applied to estimate this but it is a good approximation to think that f behaves like a random function

Analysis

- ✓ As soon as we find x_i such that $x_i=x_j$ for some j<i, we are inside the cycle modn, since $x_{i+1}=x_{j+1}$, $x_{i+2}=x_{j+2}$, κοκ

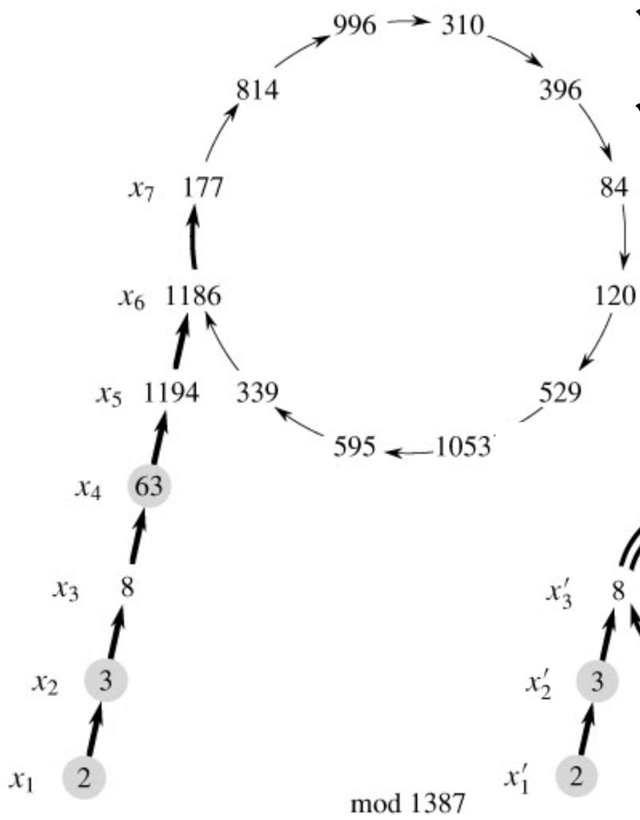


(a)

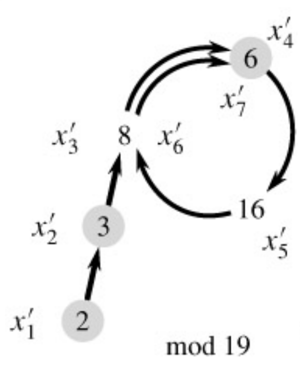
- Example: n = 1387
- $x_{i+1}=(x^2_i-1)\text{mod}1387$, with $x_1=2$.
- Factoring: 1387 =19 · 73.
- Let p be a non-trivial factor of n
- We need to identify numbers $x_i \neq x_j$ such that $x_i \equiv x_j$ modp
- Idea: as the algorithm keeps running we hope to run into a setting for y such that
  - $y \neq x_i$ modn but
  - $y \equiv x_i$ modp

Analysis

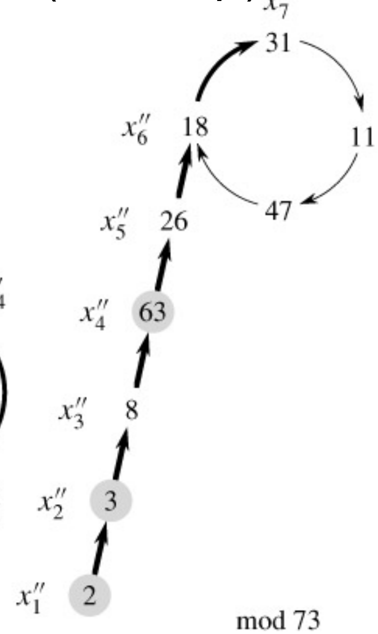
- ✓ Consider the sequence $x_i' = x_i \bmod p$ (remember we do not know p yet)
- ✓ $x'_{i+1} = x_{i+1} \bmod p = (f(x_i) \bmod n) \bmod p = f(x_i) \bmod p = ((x_i')^2 - 1) \bmod p$
- ✓ The sequence $\{x_i'\}$ also repeats itself
- ✓ The circle modp is typically much smaller than modn (about $\sqrt{p}$)



(a)



(b)



(c)

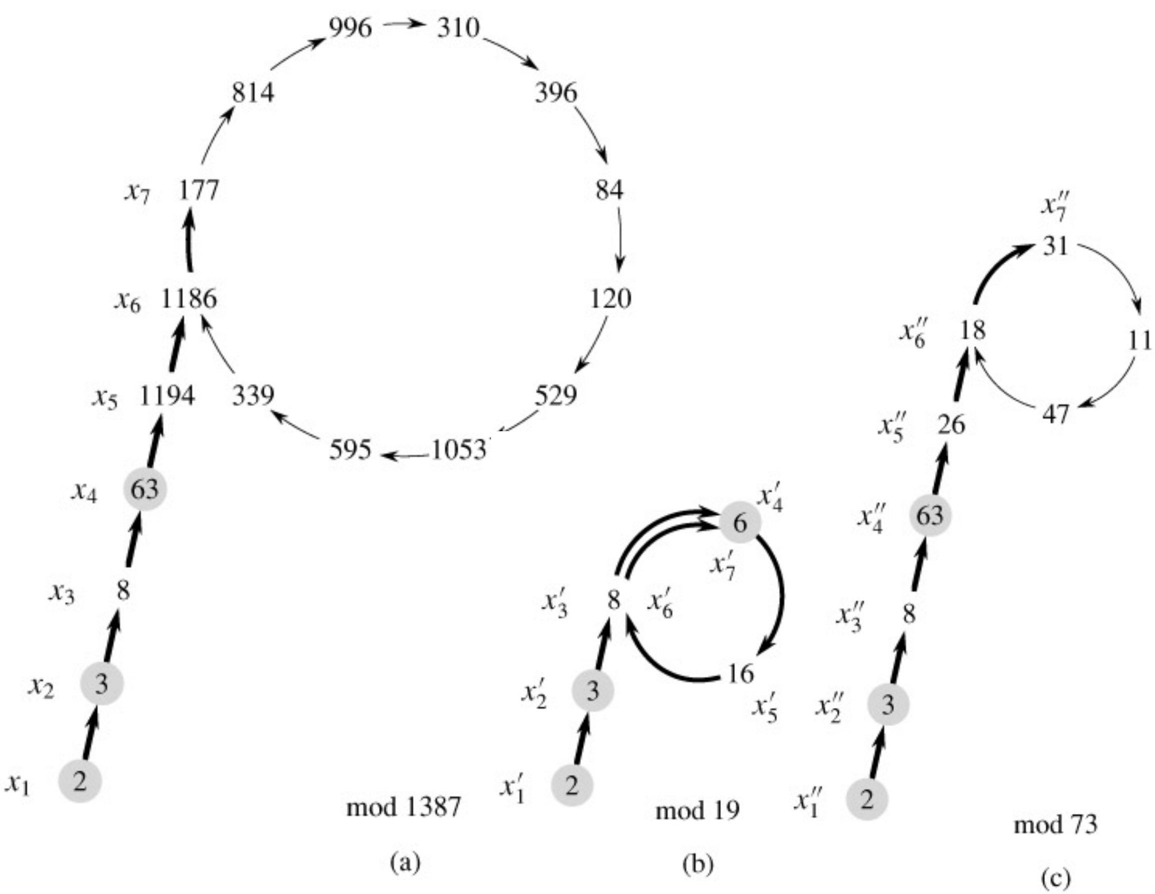
Picture (b) The cycle mod 19. Every value x_i from **(a)** is equivalent mod 19 with x'_i from **(b)**.

e.g. $x_4 = 63$ και $x_7 = 177$ are both equivalent to 6 mod 19.

Picture (c) The cycle mod 73. Every value x_i from **(a)** is equivalent mod 73, with x''_i from **(c)**.

Analysis

- ✓ Observation: once y is in the cycle modp and k is large enough, then the algorithm makes an entire loop around the cycle modp
- ✓ Hence we will check y with all other x_i values of the cycle modp.
- ✓ For one of them it will hold that $y \equiv x_i$ modp $\Rightarrow 1 < \gcd(y-x_i, n)$



Example:

$n = 1387 = 19 \cdot 73$

- ✓ The algorithm will first discover the factor 19, when we reach the point $x_7 = 177$ (it has done a loop modp)
- ✓ At that point $y = x_4 = 63$
- ✓ The algorithm will compute $\gcd(63 - 177, 1387) = 19$

- Properties of POLLARD-RHO
  - ✓ It never prints a wrong factor
  - ✓ Every integer that gets printed is a non-trivial divisor of n.
  - ✓ But there is no guarantee that it will print something
  - ✓ The running time depends on various aspects
    - The behavior of the function f(x) modn
    - The random choice we make in the beginning
    - It is also possible that if n=pq, we may keep discovering pairs x_i, x_j such that $x_i \equiv x_j$ modp and also $x_i \equiv x_j$ modq. In that case $\gcd(x_i - x_j, n) = \gcd(0, n) = n$, and no non-trivial factor is found.
  - ✓ The last issue is not really a big issue in practice
  - ✓ In practice Pollard's rho method behaves quite well (but not so well as to break RSA within a reasonable amount of time)
  - ✓ By the birthday paradox, if p is a factor of n, the cycle modp will be of length roughly $O(\sqrt{p})$
  - ✓ Since any composite number has a factor of size at most $\sqrt{n}$, it follows that on average, we expect POLLARD-RHO to produce a factor after around $O(n^{1/4})$ repetitions
  - ✓ Exponential of course since $n^{1/4} = 2^{\log n/4}$, but much better than trial division

- Other algorithms
  - ✓ Pollard's p-1 method
  - ✓ Dixon's algorithm and quadratic sieve methods
  - ✓ Methods based on elliptic curves
  - ✓ The number field sieve: the currently best theoretical worst case guarantee. It runs in time

$$e^{\left((1.92+o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}\right)}$$

  - ✓ With quantum computers, factoring can be done in polynomial time using Shor's algorithm [Shor '99]
    - But we are still far away from building quantum computers