Factorization, Discrete Logarithm and Sieve Algorithms

Brute Force

- Just try every possible solutions, maybe not every number
- If divided by 2 and found not divisible, pointless to divide by 4,6,8…
- So try all prime numbers
- Provides an upper bound on the amount of work
- Complexity is O (($2^{n/2}$) / (n/2)) where n= no. of bits
- Advantage: Always works
- No randomness in its operation
- It has a fixed upper bound

Fermat's differences of squares

- Works from the fact
- $(x+y)(x-y)=x^2- y^2$
- Take an integer N to factor
- Goal is to calculate

$$(x+y)(x-y)=x^2- y^2 =N$$

$d = x^2 - N$ (should eventually = y^2)

Algorithm

1. Set x=sqrt(N)
2. Set t= 2x + 1
3. Set d= x^2 - N (d is +ve if N is not a perfect square or 0 if N is a perfect square)

   Make t as the diff. b/n x^2 and the next square $(x+1)^2$

   t= $[(x+1)^2 - x^2]$=2x+1

|||ly $(x+2)^2$ - $(x+1)^2$ = 2x+3 which is 2 more than 2x+1

4. a. If d is a square then stop and go to Step 5
   b. Set d to d+t
   c. Set t to t+2
   d. Go to Step 4.a
5. Set x to sqrt(d+N)
6. Set y to be sqrt(d)
7. Return 2 factors x+y and x-y

Example

N=88

1. x=10
2. t=2x10+1=21
3. d= 10^2 - 88 = 12
4. a. d= 12 +21 = 33
   b. t = 21 + 2 =23
   a. d = 33 + 23 = 56
   b. t= 23 + 2=25
   a. d= 56 +21 = 81
   b. t= 25 + 2 =27
5. x = sqrt(81+88) = 13
6. y= sqrt(81)= 9
7. x+y=22 , x-y=4

Pollard Rho Factorization Algorithm

- Suppose we need we need to factorize a number N = pq, where p is a non-trivial factor
- A polynomial modulo N,

$$F(x) = x^2 + c \pmod{N}$$

is used to generate a pseudorandom sequence

- Once we choose for F a polynomial, we can easily check that for the sequence defined by

$$X_{i+1} = F(X_i) \pmod{N}$$

- reduction modulo p for any factor p of N gives the same recursion formula (modulo p instead of modulo N) for the reduced sequence
- For a prime divisor p of N, let X(p) denote the sequence X (mod p). We know that X(p) satisfies the recursion

$$X_{i+1}^{(p)} = F(X_i^{(p)}) \pmod{p}$$

Pollard Rho-Factors an integer N

1. Set b to be a random integer between 1 and N-3
2. Set s to be a random integer between 1 and N-1
3. Set A=s
4. Set B=s
5. Define f(x)= $(x^2 + b) \bmod N$
6. Let g=1
7. If g≠1 then go to Step 8
   a. Set A=f(A)
   b. Set B= f(f(B)
   c. Set g=gcd(A-B, N)
   d. Repeat (go to step 7) If g s less than N, return. Otherwise,the algorithm failed to find a factor
8. If g s less than N, return. Otherwise, the algorithm failed to find a factor

Quadratic Sieve

- Theorem: Let N be a composite natural number and X, Y be a pair of integers such that $X+Y \neq N$. If $X^2 \equiv Y^2 \pmod{N}$, then gcd(X+Y, N) and gcd(X-Y, N) are the non trivial factors of N

Quadratic Sieve Algorithm: finds factors of integer N

1. Initialization: a sequence of quadratic residues $Q(x)= (m+x)^2 - N$ is generated for small values of x where m= Floor(sqrt(N))
2. Forming the factor base : Base consists of a small collection of small primes. The set is FB={-1, 2,p1.p2,…..pt-1}
3. Sieving: Quadratic residues are now factored using the factor base. Sieving stops when t full factorizations of Q(x) have been found
4. Forming and solving the matrix: for the collection of fully factored Q(x) a matrix F is constructed. The matrix contains information about the factors. The goal of this stage is to find a linear combination of Q(x)'s which gives the quadratic congruence

Pollard's Rho-1

- Requires a list of primes upto some bound B
- Works on Fermat's Little theorem
- Given a prime p and integer b
- $b^{p-1} \equiv 1 \pmod{p}$
- Find p such that p is a prime factor of N
- If L is some multiple of (p-1) then $p \mid \gcd(b^L - 1, N)$
- To get L to be a multiple of (p-1) is to have L be a multiple of several prime factors
- Algorithm works by computing $L = p1^{e1} \ldots\ldots pk^{ek}$

- Where ei= floor((log B)/(log pi)) and trying to see if $(b^L - 1)$ and N have common factors

Pollards Rho-1: Factors an integer N , given a set of primes p1,.....pk

1. Set b=2
2. For each i in the range {1,2,.....k} perform the following steps
   a. Set e to be floor((log B)/(log pi))
   b. Set f to be pi^e
   c. Set $b=b^f$ mod N
3. Set g to be the GCD of N-1 and b
4. If g is greater than 1 and less than b, then g is a factor of b. Otherwise the algorithm fails

Elliptic Curve Factorization

Discrete Logarithms

- Pollard Rho
- Baby Step Giant Step

Sieve Algorithms-Eratosthenes's sieve

Algorithm 4.1 Eratosthenes's sieve

```
Require: Initial limit Lim
  Create array IsPrime indexed from 1 to Lim, initialized to true
  IsPrime[1] ⟵ false
  for i from 2 to Lim do
    if IsPrime[i] is true then
      j ⟵ 2i
      repeat
        IsPrime[j] ⟵ false; j ⟵ j + i
      until j > Lim
    end if
  end for
  for i from 2 to Lim do
    if IsPrime[i] is true then
      Display(i,'is prime')
    end if
  end for
```

As the runtime analysis goes, the algorithm spends $\lceil \texttt{Lim}/p \rceil$ steps in the inner loop to remove the multiples of the prime p (when i is equal to p).

Thus, the total running time is bounded by

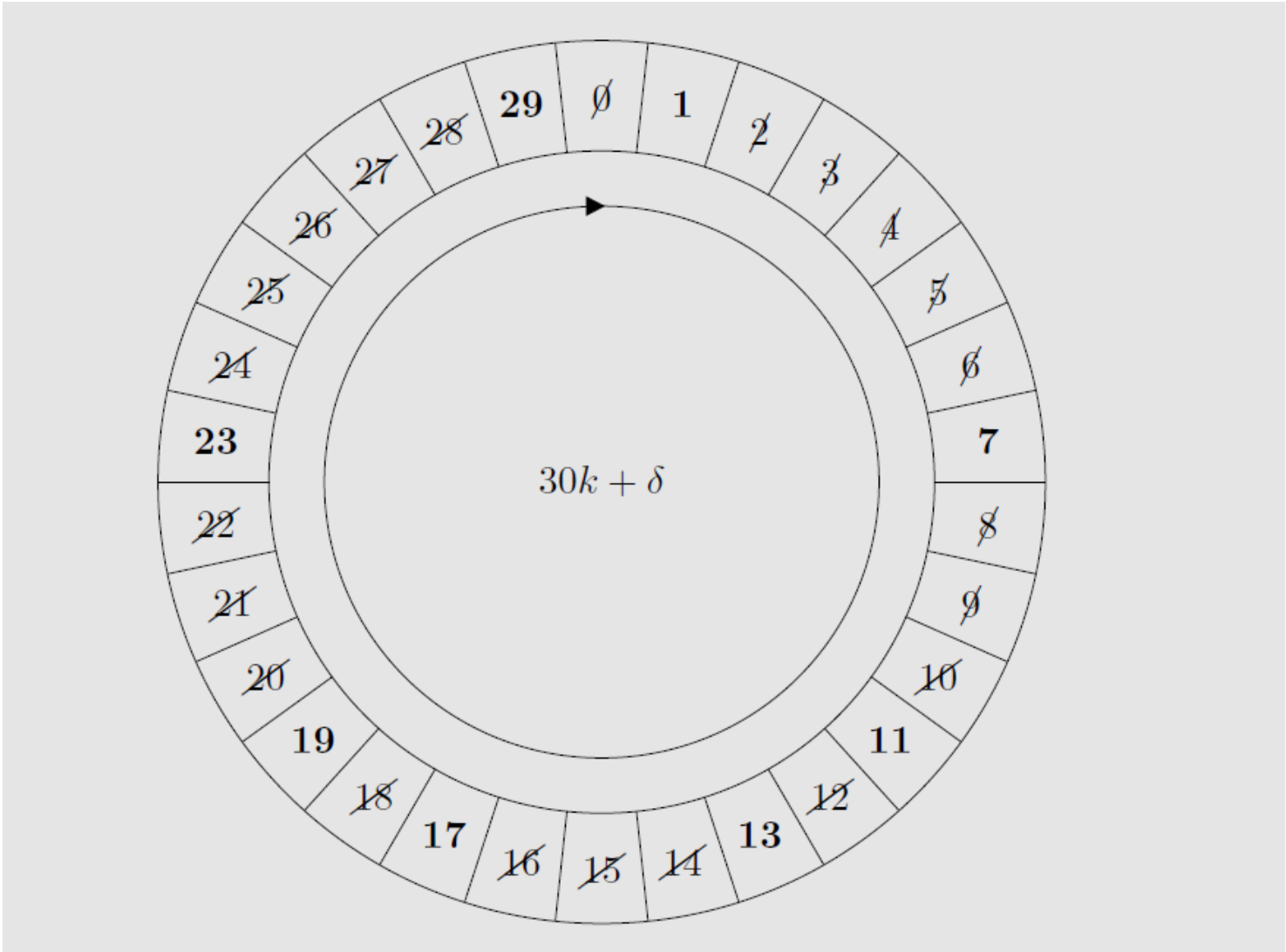
$$\sum_{p=2}^{\texttt{Lim}} \left\lceil \frac{\texttt{Lim}}{p} \right\rceil$$

Improvements to Eratosthenes's sieve

- remove all even numbers from the table in memory
- allows to run Eratosthenes's sieve twice as far for the same cost in terms of both time and memory
- consider the three primes 2, 3 and 5. Between, 1 and 30, only eight numbers, namely 1, 7, 11, 13, 17, 19, 23 and 29, are not multiples of either 2, 3 or 5
- adding 30 or a multiple of 30 to any number preserves the properties of being multiple of 2, 3 or 5
- In any interval of 30 consecutive integers, we know that Eratosthenes's sieve already removes 22 numbers, simply by sieving over the small primes 2, 3 and 5.
- to consider only numbers of the form $30k + \delta$, for all integers k and for chosen among 1, 7, 11, 13, 17, 19, 23 and 29
- Allows us to represent more than three times more numbers than the initial sieve into the same amount of memory

Schematic picture of wheel factorization

- to visualize the algorithmic principle, it is useful to represent the numbers on a modular wheel of perimeter 30



- going up to the small prime 11, 480 numbers out of 2310, thus gaining a factor of almost five
- Using wheel factorization also speeds up the computation
- drawback of complicating the matter of sieving over larger primes, since the locations of multiples of these primes are no longer evenly distributed in memory

Segmented sieve

- only primes up to square root of any composite need to be considered
- the sieving process only needs to know the prime numbers up to the initial limit
- Main part of the computation only depends on a short initial segment of Eratosthenes's sieve
- Steps:
- Divide the range 2 through n into segments of some size $\Delta \leq \sqrt{n}$.
- Find the primes in the first (i.e. the lowest) segment, using the regular sieve.
- For each of the following segments, in increasing order, with m being the segment's topmost value, find the primes in it as follows:
  - Set up a Boolean array of size Δ, and
  - Eliminate from it the multiples of each prime $p \leq \sqrt{m}$ found so far, by calculating the lowest multiple of p between $m - \Delta$ and m, and enumerating its multiples in steps of p as usual, marking the corresponding positions in the array as non-prime.

  - If Δ is chosen to be $\sqrt{n}$, the space complexity of the algorithm is $O(\sqrt{n})$, while the time complexity is the same as that of the regular sieve

- At the first segment, the smallest multiple of each sieving prime that is within the segment is calculated, then multiples of the sieving prime are marked as composite in the normal way;
- when all the sieving primes have been used, the remaining unmarked numbers in the segment are prime.
- Then, for the next segment, for each sieving prime you already know the first multiple in the current segment (it was the multiple that ended the sieving for that prime in the prior segment), so you sieve on each sieving prime, and so on until you are finished.
- Memory required to remember this location from one interval to the next

Finding primes faster: Atkin and Bernstein's sieve

- Atkin and Bernstein proposed a new algorithm, not based on Eratosthenes's sieve
- constructed from 3 subroutines each addressing a subset of the prime numbers, using a characterization of primes which does not make use of divisibility.
  - First -Used to find primes congruent to 1 modulo 4
  - second finds primes congruent to 1 modulo 6
  - third finds primes congruent to 11 modulo 12.
  - Since all primes, but 2 and 3, are of these forms these three subroutines suffice.
  - For primes congruent 1 modulo 12, we need to choose either the first or second subroutine.

THEOREM 4.1 (Th. 6.1 in [AB04])

Let n be a square-free positive integer with $n \equiv 1 \pmod 4$. Then n is prime if and only if the cardinality of the set $\{(x,y)|x>0, y>0, 4x^2+y^2=n\}$ (or equivalently of the set $\{(x,y)|x>y>0, x^2+y^2=n\}$) is odd.

THEOREM 4.2 (Th. 6.2 in [AB04])

Let n be a square-free positive integer with $n \equiv 1 \pmod 6$. Then n is prime if and only if the cardinality of the set $\{(x,y)|x>0, y>0, 3x^2+y^2=n\}$ is odd.

THEOREM 4.3 (Th. 6.3 in [AB04])

Let n be a square-free positive integer with $n \equiv 11 \pmod{12}$. Then n is prime if and only if the cardinality of the set $\{(x,y)|x>y>0, 3x^2-y^2=n\}$ is odd.

A square free integer is one which is not divisible by any perfect square other than 1

Algorithm 4.2 Sieve of Atkin and Bernstein for primes $\equiv 1 \pmod 4$

Require: Initial range $\texttt{Lim} \ldots \texttt{Lim} + 4Count$ with $\texttt{Lim} \equiv 1 \pmod 4$
  Create array $IsOdd$ indexed from 0 to $Count$, initialized to `false`.
  for all (x, y, i) with $x > 0$, $y > 0$, $i \leq Count$, $4x^2 + y^2 = \texttt{Lim} + 4i$ **do**
    Negate the Boolean value of $IsOdd[i]$
  end for
  for all prime q with $q^2 \leq \texttt{Lim} + 4Count$ **do**
    for all $\texttt{Lim} + 4i$ multiples of q^2 **do**
      Set $IsOdd[i]$ to `false`
    end for
  end for

The algorithm:

1. Create a results list, filled with 2, 3, and 5.
2. Create a sieve list with an entry for each positive integer; all entries of this list should initially be marked non prime.
3. For each entry number n in the sieve list, with modulo-sixty remainder r:
    1. If r is 1, 13, 17, 29, 37, 41, 49, or 53, flip the entry for each possible solution to $4x + y = n$
    2. If r is 7, 19, 31, or 43, flip the entry for each possible solution to $3x + y = n$.
    3. If r is 11, 23, 47, or 59, flip the entry for each possible solution to $3x – y = n$ when $x > y$.
    4. If r is something else, ignore it completely.
4. Start with the lowest number in the sieve list.
5. Take the next number in the sieve list still marked prime.
6. Include the number in the results list.
7. Square the number and mark all multiples of that square as non prime. Note that the multiples that can be factored by 2, 3, or 5 need not be marked, as these will be ignored in the final enumeration of primes.
8. Repeat steps four through seven

```cpp
// C++ program for implementation of Sieve of Atkin
#include <bits/stdc++.h>
using namespace std;

int SieveOfAtkin(int limit)
{
    // 2 and 3 are known to be prime
    if (limit > 2)
        cout << 2 << " ";
    if (limit > 3)
        cout << 3 << " ";

    // Initialise the sieve array with false values
    bool sieve[limit];
    for (int i = 0; i < limit; i++)
        sieve[i] = false;
```

```
/* Mark siev[n] is true if one
   of the following is true:
a) n = (4*x*x)+(y*y) has odd number of
   solutions, i.e., there exist
   odd number of distinct pairs (x, y)
   that satisfy the equation and
    n % 12 = 1 or n % 12 = 5.
b) n = (3*x*x)+(y*y) has odd number of
   solutions and n % 12 = 7
c) n = (3*x*x)-(y*y) has odd number of
   solutions, x > y and n % 12 = 11 */
```

```
for (int x = 1; x * x < limit; x++) {
    for (int y = 1; y * y < limit; y++) {

        // Main part of Sieve of Atkin
        int n = (4 * x * x) + (y * y);
        if (n <= limit && (n % 12 == 1 || n % 12 == 5))
            sieve[n] ^= true;

        n = (3 * x * x) + (y * y);
        if (n <= limit && n % 12 == 7)
```

```
                sieve[n] ^= true;

            n = (3 * x * x) - (y * y);
            if (x > y && n <= limit && n % 12 == 11)
                sieve[n] ^= true;
        }
    }

    // Mark all multiples of squares as non-prime
    for (int r = 5; r * r < limit; r++) {
        if (sieve[r]) {
            for (int i = r * r; i < limit; i += r * r)
                sieve[i] = false;
        }
    }
```

```
    // Print primes using sieve[]
    for (int a = 5; a < limit; a++)
        if (sieve[a])
            cout << a << " ";
}

// Driver program
int main(void)
{
    int limit = 20;
    SieveOfAtkin(limit);
    return 0;
}
```

Slides prepared by taking the contents from the text books and webresource

Antoine Joux, "*Algorithmic Cryptanalysis*", CRC Press, 2009

Christopher Swenson, "Modern Cryptanalysis", Wiley, 2008

https://www.geeksforgeeks.org/sieve-of-atkin/

By

Dr. Renuka A, Professor

Dept. of Computer Science and Engineering

Manipal Institute of Technology

Manipal

For the Course Cryptanalysis