CHAPTER 1

Exponentiation Algorithms

As hinted earlier in the course, one of the arithmetic operations that needs to be optimized for both RSA encryption, RSA signatures as well as discrete-log based systems is exponentiation. We now review various methods for fast exponentiation.

1.1 Basic Binary Ladders

The basic multiplication algorithm is based on binary ladders (we also commonly call these square-and-multiply algorithms). The point is that they work as if we are doing a recursion, except that they are still iterative.

Algorithm 1 BINEXP_LR

Require: A base x and an exponent $y = (y_{d-1} \dots y_0)_2$ in binary.
Ensure: x^y.

1: $z = x$
2: **for** $i = d-2, \dots, 0$ **do**
3: $\quad z := z^2$
4: $\quad$ **if** $y_i = 1$ **then**
5: $\quad\quad z := z \cdot x$
6: $\quad$ **end if**
7: **end for**
8: **return** z

For instance, the above algorithm will compute x^{23} as follows:

$$x^{23} = x^{2^4+2^2+2+1} = \left(((x^2)^2 \cdot x)^2 \cdot x\right)^2 \cdot x.$$

Note that if you want to do modular exponentiation (i.e., compute $x^y \bmod N$), you want to do modular multiplications in Steps 3. and 5.

This algorithm is essentially an "unrolled" version of the following simple recursive algorithm `POW` below (which you will never use in practice for large exponents - why?).

An alternative way to compute x^{23} is as follows:

$$x^{23} = x \cdot x^2 \cdot (x^2)^2 \cdot (x^8)^2$$

Algorithm 2 POW

Require: A base x and an exponent n.
Ensure: x^n.
1: **if** $n = 1$ **return** x
2: **if** n is even **return** $\mathbf{POW}(x^2, n/2)$
3: **if** n is odd **return** $\mathbf{POW}(x^2, (n-1)/2) \cdot x$

This displays slightly better the binary expansion of the number 23. The right-to-left binary ladder is then the following algorithm:

Algorithm 3 BINEXP_RL

Require: A base x and an exponent $y = (y_{d-1} \dots y_0)_2$ in binary.
Ensure: x^y.
1: $z := x$
2: $a := 1$
3: **for** $i = 0, \dots, d-1$ **do**
4: **if** $y_i = 1$ **then**
5: $a := z \cdot a$
6: **end if**
7: $z := z^2$
8: **end for**
9: **return** a

A priori, the above algorithm seems to require the same number of multiplications and squarings as Algorithm 3. There is one practical advantage of Algorithm 1 to Algorithm 3, namely, that the multiplicand x in the former is fixed and in many cases (e.g., primality testing, where x is small and one computes x^{n-1}) the multiplication $z \cdot x$ can be made quite fast.

To write the cost of the above algorithm, we assume that S is the cost of a squaring and M is the cost of multiplication. The total cost is then $(\log y)S + HM$ where H is the Hamming weight of the exponent y (i.e., the number of 1s in the binary expansion of y).

1.2 Window exponentiation

There is an exponential algorithm that is slightly more general than the binary ladder algorithms in the sense that it exploits the base-B representation of the exponent for some $B = 2^b$. It assumes that the exponents $\{x, x^2, \dots, x^{B-1}\}$ have been precomputed. The algorithm is the following:

To illustrate the advantage of the windowing ladder, consider, e.g., computing x^{79}. First, $79 = (1001111)_2 = (1033)_4$. For $B = 2^2$, the windowing ladder computes x^{79} as follows:

$$x^{79} = \left(\left(x^4\right)^4 \cdot x^3\right)^4 \cdot x^3$$

The cost of this algorithm is $6S+2M$. On the other hand, the binary ladder algorithm

Algorithm 4 WINDOWING LADDER

Require: A base x and an exponent $y = (y_{d-1} \dots y_0)_B$ in base $B = 2^b$. We assume precomputed values of $\{x^m : 0 < m < B,\ m \text{ is odd}\}$.
Ensure: x^y.

```
1: z := 1
2: for i = d − 1, ... 0 do
3:    Write y_i =: 2^c · m where m is either 0 or odd.
4:    z := (x^m)^{2^c} · z
5:    if i > 0 then
6:       z := z^{2^b}
7:    end if
8: end for
9: return z
```

computes it as follows:

$$x^{79} = \left(\left(\left(x^{2^3} \cdot x\right)^2 \cdot x\right)^2 \cdot x\right)^2 \cdot x,$$

and the cost is obviously $6S + 4M$. Of course, one should not forget the cost of the precomputation of $\{x, x^2, x^3\}$. Yet, if the exponent y is large, the windowing ladder saves multiplications.

1.3 Fixed-base ladders

The previous algorithm windowing ladder algorithm provides a hint on how to get a very efficient exponentiation algorithms when the base x is fixed (i.e., we reuse it for performing multiple exponentiations with different y's).

In this case, we can simply do a large precomputation to store all the values

$$\{x^{\ell B^j} : 1 \le \ell \le B - 1,\ 1 \le j \le d - 1\}.$$

Once this computation has been done, the algorithm then becomes very simple as all we need is a multiplication for each B-ary digit, making a total of $\dfrac{\log y}{\log B}$ multiplications.

Algorithm 5 FIXED-BASE LADDER

Require: A base x and an exponent $y = (y_{d-1} \dots y_0)_B$ in base $B = 2^b$. We assume precomputed values of $\{x^m : 0 < m < B\}$.
Ensure: x^y.

```
1: z := 1
2: for j = d − 1, ... 0 do
3:    z := x^{y_j · B^j} · z
4: end for
5: return z
```

1.4 Addition/Lucas chains/Montgomery ladder

We will see how this method works in the context of primality testing a little later in the course. For the moment, all of the previous methods motivate the following basic question:

Question 1.4.1. *What is the length of the shortest addition chain sequence $a_0, a_1, \ldots, a_r$ such that $a_0 = 1$ and $a_r = y$? Here, addition chain sequence is defined as a sequence of positive integers with the property that for every index i, there exist indices $0 \leq j, k < i$ such that $a_i = a_j + a_k$.*

The question of the length $\ell(y)$ of the shortest addition chain for the number y is an interesting one. It is difficult to compute the exact value of $\ell(y)$. Yet, Erdös showed that (see [**?**])

$$\ell(y) = \log y + (1 + o(1))\frac{\log y}{\log \log y}.$$

In practice, it is hard to compute the optimal addition chain, so one typically uses an almost optimal chain (as we see in the examples above).

Lucas chains are special types of addition chains introduced by Peter Montgomery [**?**], originally as a method of speeding up scalar multiplications on elliptic curves and subsequently, as performing exponentiations. It is an addition chain $a_0, a_1, \ldots, a_r$ such that $a_0 = 1$, for each index i, there exist indices $1 \leq j, k < i$ such that $a_i = a_j + a_k$ with either $a_j = a_k$ or $|a_j - a_k| = a_m$ for some m. These are helpful to compute recurrences of the form $X_{m+n} = f(X_m, X_n, X_{m-n})$. Montgomery also computes lower bounds on the lengths of Lucas chains in [**?**].

The concept of Lucas chains led Montgomery to propose a ladder that ended up being helpful not only for elliptic curve scalar multiplications, but also for general exponentiations. To explain the Montgomery ladder, let $y = (y_{d-1} \ldots y_0)_2$ be the binary expansion of the exponent and for each j, let $L_j = (y_{d-1} \ldots y_j)_2$. Let $H_j = L_j + 1$. Then

$$L_j = 2L_{j+1} + y_j = L_{j+1} + H_{j+1} + y_j - 1 = 2H_{j+1} + y_j - 2.$$

The idea of the Montgomery ladder is to use to registers z_0 and z_1 in which one would store x^{L_j} and x^{H_j} at any given step. Since

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) & \text{if } y_j = 0 \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) & \text{if } y_j = 1, \end{cases}$$

this leads to the following algorithm:

Even if it might seem *a priori* that the Montgomery ladder has more multiplications than the LR or RL binary ladders, it should be noted that there are three main reasons why the Montgomery ladder is of interest:

1. (Fixed R_1/R_0) - this properly was crucially used by Montgomery for speeding up elliptic curve scalar multiplication (we will see how this work later in the course, so we will not discuss it now).

Algorithm 6 MONTGOMERY LADDER

Require: A base x and an exponent $y = (y_{d-1} \dots y_0)_2$.
Ensure: x^y.

```
1: z_0 := 1, z_1 = x
2: for j = d − 1, ... 0 do
3:    if y_j = 0 then
4:       z_1 := z_0 · z_1, z_0 := (z_0)^2
5:    else
6:       z_0 = z_0 · z_1, z_1 := (z_1)^2
7:    end if
8: end for
9: return z_0
```

2. (Parallelization) - note that each pair of multiplications (in Line 4 and Line 6) can be parallelized as the multiplications are independent. For modular exponentiation, this can be an advantage - if the cost of modular multiplication is M, modular squaring is slightly faster, so the cost of each of these lines on a machine with two processors is M.

3. (Common multiplicand) - it should be noted that the two pairs of multiplications (both Lines 4 and 6) have a common multiplicand - if $y_j = b$ then the two multiplications are $z_b \cdot z_{\neg b}$ and $(z_b)^2$. This allows for some practical speedups, originally used for speeding up the above binary ladders [**?**]. The basic idea is to express the two multiplications in terms of logical operations. Since the common multiplicand in both cases is z_b, we write

$$z_{\mathtt{com}} := z_0 \wedge z_1, \qquad z_{b,c} := z_{\mathtt{com}} \oplus z_b, \qquad z_{\neg b,c} := z_{\mathtt{com}} \oplus z_{\neg b},$$

and observe that $z_{\neg b} = z_{\mathtt{com}} + z_{\neg b,c}$. We then have

$$z_b \cdot z_{\neg b} = z_b \cdot z_{\mathtt{com}} + z_b \cdot z_{\neg b,c},$$

and

$$(z_b)^2 = z_b \cdot z_{\mathtt{com}} + z_b \cdot z_{b,c}.$$

Since $z_b \cdot z_{\mathtt{com}}$ is common for the two products, it can be computed only once. The gain from the above computation comes from the fact that, on average, the Hamming weights of $z_{\mathtt{com}}$, $z_{b,c}$ and $z_{\neg b,c}$ are half the Hamming weights of the inputs, thus, the multiplications $z_b \cdot z_{b,c}$, $z_b \cdot z_{\mathtt{com}}$ and $z_b \cdot z_{\neg b,c}$ requiring half less binary additions.

1.5 Exercises

Exercise 1.1 (SAGE)**.** Here, you will do a few basic SAGE exercises related to some bit operations.

1. Write a simple one-line command (in SAGE) that calculates $3^{4324324}$ modulo

$2^{1000000}$ using the Python/SAGE generic `%` (modulo) operator. Time the calculation. You probably notice that it is quite slow. Using the Python "AND" operator `&`, show how to write another one-line command that speeds this up. Give a short justification of why you are getting the same answer. Now, implement a function `myLSB(N, k)` that takes as input integers N and k and outputs the k least significant bits in the binary representation of N.

2. Recall Python's right-shift (`» k`) and left-shift (`« k`) operators. Implement a function `myMSB(N, k)` that takes an integer N and an integer k and returns the k most significant bits of the binary representation of N.

Exercise 1.2 (SAGE)**.** Implement `Montgomery_mult` and `Montgomery_exp` corresponding to the Montgomery multiplication and exponentiation respectively. Test the correctness of your functions using the `%` operator and compare the timing.

Exercise 1.3. 1. Write a *recursive* function `Fibo_recursive(n)` (i.e., your function calls itself) that outputs the n-th Fibonacci number (1, 1, 2, 3, 5, 8, 13, ...).

2. Write an *iterative* function `Fibo_iterative(n)` (i.e., using a `for` loop) that outputs the n-th Fibonacci number.

3. Run your two functions on $n = 32$, time and compare the results.

Exercise 1.4. For an integer $n \geq 1$, let $\ell(n)$ be the shortest length of an addition chain $a_0 = 1 < a_1 < ... < a_{\ell(n)} = n$ (i.e., for every integer k such that $1 \leq k \leq \ell(n)$ there are indices $0 \leq i, j < k$ such that $a_i + a_j = a_k$). In this exercise, we are going to show that $\ell(n) \sim \log_2(n)$.

1. Show that if $2^s \leq n < 2^{s+1}$ and $s \geq 1$ then $s \leq \ell(n) \leq 2s$.

2. Prove that if $r \geq 1, s \geq 0$ are integers such that $2^{rs} \leq n < 2^{r(s+1)}$ then there is an addition chain[1] for n of length at most $(r+1)s + 2^r - 2$ and which starts with $a_i = i$ for all $i \in \{0, ..., 2^r - 2\}$, that is:

$$a_0 = 1, \;\; a_1 = 2, \;\; a_2 = 3, \;\; ..., \;\; a_{2^r-2} = 2^r - 1.$$

Hints: proceed by induction on s. In the induction step, you can work with the euclidean division of n by 2^r (and use the induction hypothesis on the quotient).

3. By choosing $r = \lceil \log(\log(n)) \rceil$ for $n \geq 3$ in b), where log = ln is the logarithm in base e, deduce that $\ell(n) \leq \log_2(n)(1 + o(1))$ as $n \to +\infty$.

Exercise 1.5. Find an addition chain $a_0 = 1 < a_1 < ... < a_{15} = 2047$ of length 15 for $n = 2047$.

[1] To be very precise for the case $s = 0$, we should say "there is an addition chain for $\max\{n, 2^r - 1\}$".