Some Algorithms for Exponentiation

Nitin Verma
mathsanew.com

July 20, 2021

Given two positive integers a and b, we need to compute a^b. Here, a is called the *Base*, b the *Exponent* and this process *Exponentiation*. A related process is *Modular Exponentiation*, where we are also given positive integer n called the *Modulus*, and need to compute $a^b \bmod n$. In this article, we will discuss some algorithms which will apply to both these processes with slight changes.

On a computer system which can handle integers only up to 64 bits, the set of integers a and b so that a^b can fit in a single hardware provided 64-bit integer is very limited. For example, to ensure $a^b < 2^{64}$ for any $a \geq 2$, b must be less than 64. For $a \geq 256\ (= 2^8)$, b must be less than 8. To perform exponentiation for larger integers a and b, we will need a software level support for handling arbitrarily large integers and arithmetic over them.

The algorithms we discuss here are applicable to any positive integers a, b and n, however large they may be. But the source code shown for them will only use int and long type of C, which are restricted in their range of integers. This allows to keep the source code simpler while we discuss the mathematics behind the algorithms. With help of a large-integers software library, this source code can be adjusted to support arbitrarily large integers.

We will be frequently using the binary representation of b. We can denote it by $(b_{m-1}b_{m-2}\ldots b_1b_0)_2$, which has m bits for some $m \geq 1$. Since $b > 0$, the most-significant-bit b_{m-1} must be 1. We will say bit b_p is at *Position* p. "MSB" and "LSB" will be used to refer to the most and least-significant-bit. Due to the way this binary form is written, leftmost and rightmost bits will mean MSB and LSB respectively. Also, a prefix (or suffix) of this binary form will mean sequence of bits starting at the MSB (or ending at the LSB).

Exponentiation vs Modular Exponentiation

All the algorithms and their C methods presented below will apply both to exponentiation and modular-exponentiation, with slight changes. The methods are shown for modular-exponentiation and so contain many uses of modulo n operation done via "`% n`", to reduce the multiplication results modulo n. Removing these "`% n`" occurrences will give us the methods for normal exponentiation.

This close similarity between the algorithms for these two exponentiations can be proven using the following relation from *Modular Arithmetic*. Here, x, y and n are any positive integers:

$$(xy) \bmod n = ((x \bmod n)(y \bmod n)) \bmod n$$

Common Structure in Algorithms

In all the algorithms we discuss below, we can observe a common structure. All of them build-up the value of a^b (modulo n) iteratively, where an integer x initialized as 0 or 1 is updated to a higher integer (or kept intact) in each iteration. The loop is terminated when x becomes b. The desired computation of a^b is performed in integer variable r, by maintaining the following loop-invariant in each iteration:

$$P\text{: } r = a^x \bmod n$$

So, when the loop terminates with $x = b$, we must have $r = a^b \bmod n$. This common structure can be expressed in pseudocode as:

```
x <- 0 (or 1)

r <- a^0 (or a^1) mod n

while(x < b)
{
  increase x to some higher integer.

  update r to maintain P for the new x.
}
```

The individual algorithms will differ in how the value of x is built-up iteratively to become b, and correspondingly how r is updated to maintain invariant P for the updated x.

We will also see that, though x helps in specifying the invariants and is critical in understanding the working of the algorithms, its presence is not really required for actual execution of the algorithms.

We will now discuss the algorithms individually.

Trivial Algorithm

A trivial algorithm is to sequentially compute $a^2, a^3, \ldots, a^{b-1}, a^b$ by multiplying an a^x term with a to obtain a^{x+1}. In the common-structure of last section, this means initializing x to 1 and increasing it by 1 in each iteration. As x is updated to $x+1$, r can be updated to (modulo n):

$$a^{x+1} = a^x a = ra$$

The following method implements this algorithm:

```
typedef unsigned int uint;
typedef unsigned long ulong;

uint trivial(uint a, uint b, uint n)
{
  uint r, x;

  x = 1;
  r = a % n;

  /* loop-invariants:
       P: r = (a ^ x) % n */

  while(x != b)
  {
    x = x + 1;
    r = ((ulong)r * a) % n; /* maintain P */
  }
  /* x = b */

  return r;
}
```

This algorithm requires $b-1$ multiplications, and so becomes inefficient as b becomes larger.

Left-to-Right Binary Algorithm

Suppose we need to compute a^{22}. We can do it by first computing a^{11} and then squaring it to give $(a^{11})^2 = a^{22}$. We can compute a^{11} similarly but since 11 is odd, we can express $a^{11} = a^{10}a$, and compute a^{10} by first computing a^5 and squaring it. This recursive process can be repeated till the trivial subproblem of computing a^1. The subproblems created in this example have exponents as (in order of their creation): 11, 5, 2, 1. This gives us a recursive algorithm based on *Divide-and-Conquer* approach. We will now try to find its iterative equivalent.

The binary representation of b helps us here. Depending upon whether any exponent e is even or odd, we are creating a subproblem with exponent $e/2$ or $(e-1)/2$. So, if the binary form of b has m bits, the first subproblem's exponent consists of $(m-1)$ prefix bits of b. By repeating this argument, we know that the subproblems (in order of their creation) will have their exponent as these many prefix bits of b: $m-1, m-2, m-3, \ldots, 2, 1$.

To create the iterative equivalent of the recursive algorithm, we can solve the subproblems in bottom to top order; that is, solving for the exponents in this order: $1, 2, 3, \ldots, m$ prefix bits of b.

In the common-structure shown earlier, x can be initialized to 1 to correspond to the single-bit prefix of b (the MSB, which is 1 here). Every iteration increases the number of prefix bits in x by 1. So, if the next bit of b to append to x is b_p, the iteration will do it by updating x to $2x + b_p$. To maintain invariant P, r will be updated to (modulo n):

$$a^{2x+b_p} = (a^x)^2 a^{b_p} = r^2 a^{b_p} = r^2 \text{ or } r^2 a \quad \{\text{for } b_p = 0 \text{ or } 1 \text{ respectively}\}$$

The loop will terminate when x contains all the m bits of b, i.e. $x = b$. The following method implements this algorithm:

```
#define MASK_POSITION31 0x80000000

uint binary_lr(uint a, uint b, uint n)
{
  uint mask = MASK_POSITION31;
  uint r, x;

  while(!(b & mask))
    mask = mask >> 1;

  /* let p denote the position of 1-bit in the mask. */
  /* p = m-1 */
  x = 1;
  r = a % n;

  /* loop-invariants:
       X: x consists of prefix bits of b from position m-1 to p.
       P: r = (a ^ x) % n */

  while(mask != 0x1)
  {
    mask = mask >> 1; /* p decreases by 1 */

    x = x * 2; /* maintain X (step 1) */
    r = ((ulong)r * r) % n; /* maintain P */

    if(b & mask)
    {
      x = x + 1; /* maintain X (step 2) */
      r = ((ulong)r * a) % n; /* maintain P */
    }
  }
  /* x = b */

  return r;
}
```

As the bits from the binary representation of b are read MSB to LSB (left to right), this algorithm is often called *Left-to-Right Binary Exponentiation* (abbreviated as "Binary LR" algorithm below).

This algorithm performs $m-1$ squarings of r, and a multiplication $r \cdot a$ for every 1 bit seen during the loop, which are maximum $m-1$. Thus, it performs total up to $2(m-1)$ multiplications, where $m = \lfloor \log_2 b \rfloor + 1$.

Right-to-Left Binary Algorithm

In the above algorithm, we iteratively read the bits of b from MSB to LSB, thus building-up x from 1 to include successively more prefix bits of b. Alternatively, we can start with x as the single-bit suffix of b (the LSB), and iteratively build-up x to include successively more suffix bits of b. Thus, bits of b will be read from LSB to MSB (right to left). For this reason, this algorithm is often called *Right-to-Left Binary Exponentiation*.

In the common-structure shown earlier, every iteration increases the number of suffix bits in x by 1. So, if the next bit of b to prepend to x is b_p (from position p), the iteration will do it by updating x to $x + b_p(2^p)$. To maintain invariant P, r will be updated to (modulo n):

$$a^{x+b_p(2^p)} = a^x(a^{2^p})^{b_p} = r(a^{2^p})^{b_p} = r \text{ or } ra^{2^p} \quad \{\text{for } b_p = 0 \text{ or } 1 \text{ respectively}\}$$

A separate variable y will be maintaining the value of a^{2^p} in it as each iteration increments p by 1, starting from $p = 0$. The following method implements this algorithm:

```
uint binary_rl(uint a, uint b, uint n)
{
  uint mask, msb_mask = MASK_POSITION31;
  uint r, x, y;

  while(!(b & msb_mask))
    msb_mask = msb_mask >> 1;

  mask = 0x1;
  /* let p denote the position of 1-bit in the mask. */
  /* p = 0 */
  x = (b & mask);
  y = a % n;
  r = (x == 1 ? a % n : 1); /* x is 1 or 0 */

  /* loop-invariants:
       X: x consists of suffix bits of b from position p to 0.
       P: r = (a ^ x) % n
       Y: y = (a ^ (2^p)) % n */

  while(mask != msb_mask)
  {
    mask = mask << 1; /* p increases by 1 */

    y = ((ulong)y * y) % n; /* maintain Y */

    if(b & mask)
    {
      x = x | mask; /* effectively adding 2^p to x, to maintain X */
      r = ((ulong)r * y) % n; /* maintain P */
    }
  }
  /* x = b */

  return r;
}
```

This algorithm performs $m-1$ squarings of y, and a multiplication $r \cdot y$ for every 1 bit seen during the loop, which are maximum $m-1$. Thus, it performs total up to $2(m-1)$ multiplications, where $m = \lfloor \log_2 b \rfloor + 1$.

2^k-ary Algorithm

In the Binary LR algorithm, we multiply r with a whenever the bit read in an iteration is 1. Now suppose $r = r_1$ just before an iteration. The

upcoming 4 iterations will be reading 4 bits from b (assuming these many bits are left), say $(1011)_2$, while updating r as:

$$r = r_1^2 a, \quad r = r^2, \quad r = r^2 a, \quad r = r^2 a$$

Thus the value of r after these 4 iterations will be: $r_1^{16} a^{11} = r_1^{2^4} a^{(1011)_2}$. Note that this required 4 squarings of r, and 3 multiplications $r \cdot a$ (as there are 3 1-bit in $(1011)_2$).

In many applications of exponentiation, b may contain a large number of bits, say, in the order of 1000. Suppose, the pattern of 4 bits $(1011)_2$ is seen again in later iterations. Then again we will be doing the above mentioned 4 squarings and 3 multiplications. But suppose we had computed $a^{(1011)_2}$ once and saved it somewhere. Then, whenever we find 4 consecutive bits of b being $(1011)_2$, we can update r directly to r^{2^4} (using 4 squarings: $r^{2^1}, r^{2^2}, r^{2^3}, r^{2^4}$) and then to $r^{2^4} a^{(1011)_2}$ (using 1 multiplication with the saved value). Thus, using the precomputed value, we could reduce the number of multiplications from 4+3 to 4+1 for every occurrence of the pattern $(1011)_2$ in b. Though, such precomputations will also be requiring certain multiplications, which we need to account for.

The 2^k-*ary Algorithm* is based on the above idea. The bits in b are considered in groups of k ($k \geq 2$) consecutive bits; the least significant k bits form a group and so on. An optimal value of k is chosen for this. Each group of k bits forms a number in range 0 to $2^k - 1$, and can be treated as a digit d in base 2^k representation of b. The values a^d are precomputed and stored in a table for each digit d in base 2^k (as we will see, we need not do it for all digits). The algorithm proceeds similarly to Binary LR algorithm.

In the common-structure shown earlier, x will contain some prefix digits (in base 2^k) of b at any point. It is initialized to 0 to correspond to 0 prefix digits of b. Every iteration increases the number of prefix digits in x by 1. So, if the next digit of b to append to x is d, the iteration will do it by updating x to $(2^k)x + d$. To maintain invariant P, r can be updated to (modulo n):

$$a^{(2^k)x+d} = (a^x)^{2^k} a^d = r^{2^k} a^d$$

where a^d is a precomputed value.

Now we will see how the number of values to precompute can be reduced. Any nonzero digit d in base 2^k can be written:

$$d = 2^t o, \quad \text{where } 0 \leq t < k,\ o \in \{1, 3, 5, \ldots, 2^k - 1\}$$

(t represents "power of two", and o "odd"). We will precompute only the odd powers of a: $a^1, a^3, a^5, \ldots, a^{2^k-1}$. In each iteration when x is updated to $(2^k)x + d$, to maintain invariant P, r will be updated to (modulo n):

$$a^{(2^k)x+d} = (a^x)^{2^k} a^{2^t o} = r^{2^k} a^{2^t o} = (r^{2^{k-t}} a^o)^{2^t}$$

For this, first $r^{2^{k-t}}$ will be computed using $(k-t)$ squarings of r. Then, the precomputed a^o will be multiplied with the updated r, and this result itself will be squared t times.

Note that when d is 0, we can simplify the update to r as:

$$a^{(2^k)x+d} = (a^x)^{2^k} a^0 = r^{2^k}$$

Following is an implementation of this algorithm:

```
void precompute(uint *table, uint tsize, uint a, uint n)
{
  int i;
  uint aa = ((ulong)a * a) % n;

  /* table[i] will contain a^(2i+1) % n */

  i = 0;
  table[0] = a % n;

  while(i < tsize-1)
  {
    i = i + 1;
    table[i] = ((ulong)table[i-1] * aa) % n;
  }
}

void split_d(uint d, uint *t, uint *o)
{
  *t = 0;

  /* loop-invariant: d * 2^t = (initial d) */

  while((d & 0x1) == 0)
  {
    d = d >> 1;
    (*t)++;
  }

  *o = d;
}
```

```
void setup_mask(ulong *mask1, int *p1, uint b, int k)
{
  ulong mask;
  int p;

  mask = (1 << k) - 1;

  /* mask will always have some consecutive k bits as 1. Currently,
     it has the least-significant k bits as 1. We will need to read
     base 2^k digits from b, where each digit is a group of k bits.
     We now shift this k-bit mask so that it coincides with the
     most-significant 2^k digit (k bits) in b. */

  /* Number of base 2^k digits in 32 bits are (32 + k-1)/k. */
  p = ((32 + k-1)/k - 1)*k;
  mask = mask << p;

  while(!(b & mask))
  {
    mask = mask >> k;
    p = p - k;
  }

  *mask1 = mask;
  *p1 = p;
}

uint base2k_ary(uint a, uint b, uint n)
{
  /* although k = 3 here, we should choose optimal k based on m */
  int k = 3, i, p;
  uint B, *table, d, t, o;
  ulong mask;
  uint r, x;

  B = 1 << k; /* 2^k */

  table = malloc((B/2)*sizeof(uint));

  precompute(table, B/2, a, n);

  /* p tracks the position of the rightmost (k-th) 1-bit in mask */
  setup_mask(&mask, &p, b, k);

  x = 0;
  r = 1;
```

```
  /* loop-invariants:
       X: x consists of prefix bits of b from position m-1 to p+k.
       P: r = (a ^ x) % n */

  while(mask != 0)
  {
    /* read the base 2^k digit from the mask position */
    d = (b & mask) >> p;

    x = x*B + d; /* maintain X corresponding to update in step M */

    /* x was updated, now update r to maintain P */
    if(d > 0)
    {
      split_d(d, &t, &o);
      /* d = o * 2^t */

      /* compute r^(2^(k-t)) */
      for(i = 0; i < k-t; i++)
        r = ((ulong)r * r) % n;

      /* compute r*(a^o) */
      r = ((ulong)r * table[o/2]) % n;

      /* compute r^(2^t) */
      for(i = 0; i < t; i++)
        r = ((ulong)r * r) % n;
    }
    else
    {
      /* compute r^(2^k) */
      for(i = 0; i < k; i++)
        r = ((ulong)r * r) % n;
    }

    mask = mask >> k;  /* step M */
    p = p - k;
  }
  /* x = b */

  free(table);

  return r;
}
```

In each iteration except the first one, this algorithm performs total k squarings of r. The first iteration starts with $r = 1$, and so the first (inner) loop of $k-t$ squarings of r has negligible cost, leaving us with only the second

loop of t squarings of r. The total number of iterations equals the number of base 2^k digits in the m bits of b, i.e. $\lceil m/k \rceil$. The total squarings of r will come out to be maximum $m-1$, which is same as the $m-1$ squarings of r in Binary LR algorithm.

Also, each iteration does 1 multiplication $r \cdot a^o$ if $d \neq 0$. So, there will be up to $\lceil m/k \rceil$ such multiplications by all iterations. The precomputation itself involves 1 squaring (a^2) and $2^{k-1}-1$ multiplications, hence total 2^{k-1} multiplications.

Thus, maximum total number of multiplications (including those for squaring) required is:

$$(m-1) + \left\lceil \frac{m}{k} \right\rceil + 2^{k-1} \approx m + \frac{m}{k} + 2^{k-1} \tag{1}$$

For a given m, let N_b and $N(k)$ (for any integer k) denote the number of multiplications required by Binary LR algorithm and 2^k-ary algorithm respectively. For $k = 2$, $N(k) < N_b$ iff:

$$\begin{aligned} & m + \frac{m}{2} + 2^{2-1} < 2(m-1) \\ \Leftrightarrow \quad & m > 8 \end{aligned} \tag{2}$$

Similarly, for $k = 3$ and $k = 4$ also, when $m \leq 8$, $N(k)$ exceeds N_b. So, for the case of $m \leq 8$, we can prefer Binary LR algorithm over 2^k-ary algorithm. Now we will see how to choose an optimal k when $m > 8$.

2^k-ary Algorithm: Optimal k

We can choose the value of integer k ($k \geq 2$) which minimizes the number of multiplications in (1) for a given m, which means to minimize:

$$f(k) = \frac{m}{k} + 2^{k-1}$$

For $m > 8$, as k increases from 0 to higher integers, $f(k)$ first keeps decreasing and then keeps increasing. So, to find integer k where $f(k)$ is minimum, we will look for the integer k at which this $f(k)$ increase starts. More precisely, we need the smallest integer k satisfying:

$$\begin{aligned} & f(k) \leq f(k+1) \\ \Leftrightarrow \quad & \frac{m}{k} + 2^{k-1} \leq \frac{m}{k+1} + 2^k \\ \Leftrightarrow \quad & \frac{m}{k(k+1)} \leq 2^{k-1} \\ \Leftrightarrow \quad & m \leq 2^{k-1}k(k+1) \end{aligned}$$

The below table shows some ranges of m and their optimal k value:

m	9–12	13–48	49–160	161–480	481–1344	1345–3584	3585–9216
k	2	3	4	5	6	7	8

Consider some $m > 8$, and let k_o denote the corresponding optimal k. Then, $N(k_o)$ must not exceed $N(k)$ for any integer $k \geq 2$. So, $N(k_o) \leq N(2)$. But, as seen in (2), for all $m > 8$, $N(2) < N_b$. Thus,

$$N(k_o) < N_b$$

In other words, for all $m > 8$, the number of multiplications required by 2^k-ary algorithm with optimally chosen k is always less than that required by Binary LR algorithm.

The number of multiplications saved by this algorithm, compared to the Binary LR algorithm, become more significant as m increases. For example, with $m = 18$ and $m = 52$, the Binary LR algorithm requires $(2(m-1) =)$ 34 and 102 multiplications, but this algorithm requires (due to (1)) $18 + (18/3) + 2^2 = 28$ $(k = 3)$ and $52 + (52/4) + 2^3 = 73$ $(k = 4)$ multiplications respectively.

■