Last update: July 27, 2025 Translated From: e-maxx.ru

Binary Exponentiation

Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate a^n using only $O(\log n)$ multiplications (instead of $O(n)$ multiplications required by the naive approach).

It also has important applications in many tasks unrelated to arithmetic, since it can be used with any operations that have the property of **associativity**:

$$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$$

Most obviously this applies to modular multiplication, to multiplication of matrices and to other problems which we will discuss below.

Algorithm

Raising a to the power of n is expressed naively as multiplication by a done $n - 1$ times: $a^{n} = a \cdot a \cdot \ldots \cdot a$. However, this approach is not practical for large a or n.

$a^{b+c} = a^b \cdot a^c$ and $a^{2b} = a^b \cdot a^b = (a^b)^2$.

The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

Let's write n in base 2, for example:

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Since the number n has exactly $\lfloor \log_2 n \rfloor + 1$ digits in base 2, we only need to perform $O(\log n)$ multiplications, if we know the powers $a^1, a^2, a^4, a^8, \dots, a^{2^{\lfloor \log n \rfloor}}$.

So we only need to know a fast way to compute those. Luckily this is very easy, since an element in the sequence is just the square of the previous element.

$$\begin{align}
3^1 &= 3 \\
3^2 &= \left(3^1\right)^2 = 3^2 = 9 \\
3^4 &= \left(3^2\right)^2 = 9^2 = 81 \\
3^8 &= \left(3^4\right)^2 = 81^2 = 6561
\end{align}$$

So to get the final answer for 3^{13}, we only need to multiply three of them (skipping 3^2 because the corresponding bit in n is not set): $3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$

The final complexity of this algorithm is $O(\log n)$: we have to compute $\log n$ powers of a, and then have to do at most $\log n$ multiplications to get the final answer from them.

The following recursive approach expresses the same idea:

$$a^n = \begin{cases}
1 &\text{if } n == 0 \\
\left(a^{\frac{n}{2}}\right)^2 &\text{if } n > 0 \text{ and } n \text{ even}\\
\left(a^{\frac{n - 1}{2}}\right)^2 \cdot a &\text{if } n > 0 \text{ and } n \text{ odd}\\
\end{cases}$$

Implementation

First the recursive approach, which is a direct translation of the recursive formula:

```
long long binpow(long long a, long long b) {
    if (b == 0)
        return 1;
    long long res = binpow(a, b / 2);
    if (b % 2)
        return res * res * a;
    else
        return res * res;
}
```

The second approach accomplishes the same task without recursion. It computes all the powers in a loop, and multiplies the ones with the corresponding set bit in n. Although the complexity of both approaches is identical, this approach will be faster in practice since we don't have the overhead of the recursive calls.

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

Applications

Effective computation of large exponents modulo a number

Problem: Compute $x^n \bmod m$. This is a very common operation. For instance it is used in computing the modular multiplicative inverse.

Solution: Since we know that the modulo operator doesn't interfere with multiplications ($a \cdot b \equiv (a \bmod m) \cdot (b \bmod m) \pmod m$), we can directly use the same code, and just replace every multiplication with a modular multiplication:

If you take two numbers and multiply them, and then take modulo m at the end, you'll get the same answer as if you took modulo m before multiplying.

```
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

Note: It's possible to speed this algorithm for large $b >> m$. If m is a prime number $x^n \equiv x^{n \bmod (m-1)} \pmod{m}$ for prime m, and $x^n \equiv x^{n \bmod{\phi(m)}} \pmod{m}$ for composite m. This follows directly from Fermat's little theorem and Euler's theorem, see the article about Modular Inverses for more details.

Effective computation of Fibonacci numbers

Problem: Compute n-th Fibonacci number F_n.

Solution: For more details, see the Fibonacci Number article. We will only go through an overview of the algorithm. To compute the next Fibonacci number, only the two previous ones are needed, as $F_n = F_{n-1} + F_{n-2}$. We can build a 2×2 matrix that describes this transformation: the transition from F_i and F_{i+1} to F_{i+1} and F_{i+2}. For example, applying this transformation to the pair F_0 and F_1 would change it into F_1 and F_2. Therefore, we can raise this transformation matrix to the n-th power to find F_n in time complexity $O(\log n)$.

Applying a permutation k times

Problem: You are given a sequence of length n. Apply to it a given permutation k times.

Solution: Simply raise the permutation to k-th power using binary exponentiation, and then apply it to the sequence. This will give you a time complexity of $O(n \log k)$.

```
vector<int> applyPermutation(vector<int> sequence, vector<int> permutation) {
    vector<int> newSequence(sequence.size());
    for(int i = 0; i < sequence.size(); i++) {
        newSequence[i] = sequence[permutation[i]];
    }
    return newSequence;
}

vector<int> permute(vector<int> sequence, vector<int> permutation, long long k) {
    while (k > 0) {
        if (k & 1) {
            sequence = applyPermutation(sequence, permutation);
        }
        permutation = applyPermutation(permutation, permutation);
        k >>= 1;
    }
    return sequence;
}
```

Note: This task can be solved more efficiently in linear time by building the permutation graph and considering each cycle independently. You could then compute k modulo the size of the cycle and find the final position for each number which is part of this cycle.

Fast application of a set of geometric operations to a set of points

Problem: Given n points p_i, apply m transformations to each of these points. Each transformation can be a shift, a scaling or a rotation around a given axis by a given angle. There is also a "loop" operation which applies a given list of transformations k times ("loop" operations can be nested). You should apply all transformations faster than $O(n \cdot length)$, where $length$ is the total number of transformations to be applied (after unrolling "loop" operations).

Solution: Let's look at how the different types of transformations change the coordinates:

- Shift operation: adds a different constant to each of the coordinates.
- Scaling operation: multiplies each of the coordinates by a different constant.
- Rotation operation: the transformation is more complicated (we won't go in details here), but each of the new coordinates still can be represented as a linear combination of the old ones.

As you can see, each of the transformations can be represented as a linear operation on the coordinates. Thus, a transformation can be written as a 4×4 matrix of the form:

$$\begin{pmatrix}
a_{11} & a_ {12} & a_ {13} & a_ {14} \\
a_{21} & a_ {22} & a_ {23} & a_ {24} \\
a_{31} & a_ {32} & a_ {33} & a_ {34} \\
a_{41} & a_ {42} & a_ {43} & a_ {44}
\end{pmatrix}$$

that, when multiplied by a vector with the old coordinates and a unit gives a new vector with the new coordinates and a unit:

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot
\begin{pmatrix}
a_{11} & a_ {12} & a_ {13} & a_ {14} \\
a_{21} & a_ {22} & a_ {23} & a_ {24} \\
a_{31} & a_ {32} & a_ {33} & a_ {34} \\
a_{41} & a_ {42} & a_ {43} & a_ {44}
\end{pmatrix}
 = \begin{pmatrix} x' & y' & z' & 1 \end{pmatrix}$$

(Why introduce a fictitious fourth coordinate, you ask? That is the beauty of homogeneous coordinates, which find great application in computer graphics. Without this, it would not be possible to implement affine operations like the shift operation as a single matrix multiplication, as it requires us to *add* a constant to the coordinates. The affine transformation becomes a linear transformation in the higher dimension!)

Here are some examples of how transformations are represented in matrix form:

- Shift operation: shift x coordinate by 5, y coordinate by 7 and z coordinate by 9.

$$\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
5 & 7 & 9 & 1
\end{pmatrix}$$

- Scaling operation: scale the x coordinate by 10 and the other two by 5.

$$\begin{pmatrix}
10 & 0 & 0 & 0 \\
0 & 5 & 0 & 0 \\
0 & 0 & 5 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}$$

- Rotation operation: rotate θ degrees around the x axis following the right-hand rule (counter-clockwise direction).

$$\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}$$

Now, once every transformation is described as a matrix, the sequence of transformations can be described as a product of these matrices, and a "loop" of k repetitions can be described as the matrix raised to the power of k (which can be calculated using binary exponentiation in $O(\log{k})$). This way, the matrix which represents all transformations can be calculated first in $O(m \log{k})$, and then it can be applied to each of the n points in $O(n)$ for a total complexity of $O(n + m \log{k})$.

Number of paths of length k in a graph

Problem: Given a directed unweighted graph of n vertices, find the number of paths of length k from any vertex u to any other vertex v.

Solution: This problem is considered in more detail in a separate article. The algorithm consists of raising the adjacency matrix M of the graph (a matrix where $m_{ij} = 1$ if there is an edge from i to j, or 0 otherwise) to the k-th power. Now m_{ij} will be the number of paths of length k from i to j. The time complexity of this solution is $O(n^3 \log k)$.

Note: In that same article, another variation of this problem is considered: when the edges are weighted and it is required to find the minimum weight path containing exactly k edges. As shown in that article, this problem is also solved by exponentiation of the adjacency matrix. The matrix would have the weight of the edge from i to j, or ∞ if there is no such edge. Instead of the usual operation of multiplying two matrices, a modified one should be used: instead of multiplication, both values are added, and instead of a summation, a minimum is taken. That is: $result_{ij} = \min\limits_{1\ \leq\ k\ \leq\ n}(a_{ik} + b_{kj})$.

Variation of binary exponentiation: multiplying two numbers modulo m

Problem: Multiply two numbers a and b modulo m. a and b fit in the built-in data types, but their product is too big to fit in a 64-bit integer. The idea is to compute $a \cdot b \pmod m$ without using bignum arithmetics.

Solution: We simply apply the binary construction algorithm described above, only performing additions instead of multiplications. In other words, we have "expanded" the multiplication of two numbers to $O (\log m)$ operations of addition and multiplication by two (which, in essence, is an addition).

$$a \cdot b = \begin{cases}
0 &\text{if }a = 0 \\
2 \cdot \frac{a}{2} \cdot b &\text{if }a > 0 \text{ and }a \text{ even} \\
2 \cdot \frac{a-1}{2} \cdot b + b &\text{if }a > 0 \text{ and }a \text{ odd}
\end{cases}$$

Note: You can solve this task in a different way by using floating-point operations. First compute the expression $\frac{a \cdot b}{m}$ using floating-point numbers and cast it to an unsigned integer q. Subtract $q \cdot m$ from $a \cdot b$ using unsigned integer arithmetics and take it modulo m to find the answer. This solution looks rather unreliable, but it is very fast, and very easy to implement. See here for more information.

Practice Problems

- UVa 1230 - MODEX
- UVa 374 - Big Mod
- UVa 11029 - Leading and Trailing
- Codeforces - Parking Lot
- leetcode - Count good numbers
- Codechef - Chef and Riffles
- Codeforces - Decoding Genome
- Codeforces - Neural Network Country
- Codeforces - Magic Gems
- SPOJ - The last digit
- SPOJ - Locker
- LA - 3722 Jewel-eating Monsters
- SPOJ - Just add it
- Codeforces - Stairs and Lines

Contributors:
jakobkogler (41.95%) gabrielsimoes (17.98%) tcNickolas (11.24%) back2sqr1 (5.99%) RodionGork (5.62%) Electron1997 (3.0%) adamant-pwn (2.62%) shark_s (2.25%) Morass (1.87%) turfaa (1.87%) roundspecs (1.5%) wangwillian0 (0.75%) arjunUpatel (0.37%) snymm2 (0.37%) PAndaContron (0.37%) rskbansal (0.37%) zhangdavid275 (0.37%) ahmedibrahim404 (0.37%) wikku (0.37%) algmyr (0.37%) vatsalsharma376 (0.37%)