# GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: EXPONENCIACIÓN BINARIA

# 1. Introducción

Digamos que tenemos la necesidad de calcular la operación $A^N$ en la cual no podemos utilizar la función de potenciación habitual del lenguaje bien porque no es conveniente o por ejemplo $A$ es una matriz. Una idea trivial sería realizar $N$ multiplicaciones de $A$ lo cual sería un algoritmo con una complejidad de $O(n)$. Pero existirá algo más eficiente para calcular $A^N$ ?.

# 2. Conocimientos previos

## 2.1. AND lógico & (palabra clave bitand)

Este operador binario compara ambos operandos bit a bit, y como resultado devuelve un valor construido de tal forma, que cada bits es 1 si los bits correspondientes de los operandos están a 1. En caso contrario, el bit es 0.

Sintaxis:

$AND - expresion\&equality - expresion$

Ejemplo:

```
int x = 10, y = 20;//x en binario es 01010, y en binario 10100

int z = x & y;//equivale a: int z = x bitand y; z toma el valor 00000 es decir 0
```

## 2.2. Desplazamiento a izquierda $<<$

Este operador binario realiza un desplazamiento de bits a la izquierda. El bit más significativo (más a la izquierda) se pierde, y se le asigna un 0 al menos significativo (el de la derecha). El operando derecho indica el número de desplazamientos que se realizarán. Los desplazamientos no son rotaciones; los bits que salen por la izquierda se pierden, los que entran por la derecha se rellenan con ceros. Este tipo de desplazamientos se denominan lógicos en contraposición a los cíclicos o rotacionales.

Sintaxis:

$expr - desplazada << expr - desplazamiento$

El patrón de bits de expr-desplazada sufre un desplazamiento izquierda del valor indicado por la expr-desplazamiento. Ambos operandos deben ser números enteros o enumeraciones. En caso contrario, el compilador realiza una conversión automática de tipo. El resultado es del tipo del primer operando. expr-desplazamiento, una vez promovido a entero, debe ser un entero positivo y menor que la longitud del primer operando. En caso contrario el resultado es indefinido (depende de la implementación).

Ejemplo:

```
unsigned long x = 10; // En binario el 10 es 1010
```

```
int y = 2;

unsigned long z = x << y;
/* z tienen el valor 40 que en binario es 101000 vea que se adiciono dos ceros al
    final que son los desplazamientos.*/
```

## 3. Desarrollo

La generalización más obvia: los restos de un determinado módulo (obviamente, la asociatividad se conserva). Lo siguiente en la *popularidad* es una generalización del producto matricial (es una asociatividad bien conocida).

Tenga en cuenta que para cualquier número de una identidad obvia factible de un número par (que se deduce de la asociatividad de la multiplicación):

$a^n = (a^{n/2})^2 = a^{n/2} * a^{n/2}$

Es el principal método de exponenciación binaria. De hecho, incluso para n Hemos demostrado cómo, después de realizar solo una operación de multiplicación, podemos reducir el problema a menos de la mitad de la potencia. Queda por entender qué hacer si el grado de n es impar. Aquí lo que hacemos es muy simple: ve en la medida n-1 Eso tendrá incluso:

$a^n = a^{n-1} * a$

Entonces, en realidad encontramos una fórmula recursiva: el grado de n que vamos, si es igual a n/2 y de lo contrario, a n-1. Está claro que no habrá más transiciones 2 $\log(n)$ antes de que lleguemos a n = 0 (basado en la fórmula recursiva).

## 4. Implementación

### 4.1. C++

Variante recursiva:

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

Variante iterativa (la división por 2 se reemplaza por operaciones de bit):

```
int binpow (int a, int n){
   int res = 1;
```

```
    while (n){
        if(n&1) res*= a;
        a*=a;
        n>>=1;
    }
    return res;
}
```

### 4.2. Java

Variante recursiva:

```
public int binpow (int a, int n){
    if(n == 0) return 1;
    if(n%2 == 1) return binpow (a, n - 1) * a;
    else {
        int b = binpow (a, n / 2);
        return b * b;
    }
}
```

Variante iterativa (la división por 2 se reemplaza por operaciones de bit):

```
public int binpow (int a, int n){
    int res = 1;
    while (n){
        if(n&1) res*= a;
        a*=a;
        n>>=1;
    }
    return res;
}
```

## 5. Complejidad

La exponenciaión binaria es una técnica que permite generar cualquier cantidad de potencia $n^{th}$ para multiplicaciones O ($\log N$) (en lugar de n multiplicaciones en el método habitual).

## 6. Aplicaciones

La técnica descrita aquí es aplicable a cualquier operación asociativa, no solo a la multiplicación de números. Recordar que la operación se llama asociativa, si para alguna a, b, c se lleva a cabo: (a*b)*c= a*(b*c). El algoritmo aquí descrito entra entre los ejemplos de que cumplen con la idea algorítmica de divide y vencerás.

A continuación vamos a mencionar algunas aplicaciones de esta técnica:

- **Cálculo efectivo de grandes exponentes módulo a número:** Calcular $x^n \bmod m$. Esta es una operación muy común. Por ejemplo, se utiliza para calcular el inverso multiplicativo modular. Como sabemos que el operador de módulo no interfiere con las multiplicaciones ($a \cdot b \equiv (a \bmod m) \cdot (b \bmod m) \pmod m$), podemos usar directamente el mismo código y simplemente reemplazar cada multiplicación con una multiplicación modular.
- **Cálculo efectivo de los números de Fibonacci:** Calcular $n$-ésimo número de Fibonacci $F_n$. Para calcular el siguiente número de Fibonacci, solo se necesitan los dos anteriores, ya que $F_n = F_{n-1} + F_{n-2}$. Podemos construir un $2 \times 2$ matriz que describe esta transformación: la transición de $F_i$ y $F_{i+1}$ a $F_{i+1}$ y $F_{i+2}$ . Por ejemplo, aplicando esta transformación al par $F_0$ y $F_1$ lo cambiaria por $F_1$ y $F_2$. Por lo tanto, podemos elevar esta matriz de transformación a la $n$-ésima potencia para encontrar $F_n$ en la complejidad del tiempo $O(\log n)$.
- **Aplicar una permutación $k$ veces:** Te dan una secuencia de longitud $n$. Aplicarle una permutación dada $k$ veces. Simplemente eleva la permutación a $k$-ésima potencia usando exponenciación binaria, y luego aplicarla a la secuencia. Esto le dará una complejidad de tiempo de $O(n \log k)$.
- **Aplicación rápida de un conjunto de operaciones geométricas a un conjunto de puntos:** Dado $n$ puntos $p_i$ , aplicar $m$ transformaciones a cada uno de estos puntos. Cada transformación puede ser un cambio, una escala o una rotación alrededor de un eje dado por un ángulo dado. También hay una operación de "bucle"que aplica una lista dada de transformaciones $k$ veces (las operaciones de "bucle"se pueden anidar). Debe aplicar todas las transformaciones más rápido que $O(n \cdot longitud)$, dónde $longitud$ es el número total de transformaciones que se aplicarán (después de desenrollar las operaciones de "bucle").
- **Números de caminos de longitud K en un grafo:** Dada un grafo no ponderada dirigida de $n$ vértices, encuentre el número de caminos de longitud $k$ desde cualquier vértice $u$ a cualquier otro vértice $v$. El algoritmo consiste en elevar la matriz de adyacencia $M$ del grafo (una matriz donde $m_{ij} = 1$ si hay una arista de $i$ a $j$, o $0$ de lo contrario) a la $k$-ésima potencia. Ahora $m_{ij}$ será el número de caminos de longitud $k$ de $i$ a $j$. La complejidad temporal de esta solución es $O(n^3 \log k)$.
- **Variación de la exponenciación binaria: multiplicar dos números módulo $m$:** Multiplicar dos números $a$ y $b$ módulo $m$. $a$ y $b$ caben en los tipos de datos incorporados, pero su producto es demasiado grande para caber en un entero de 64 bits. La idea es calcular $a \cdot b \pmod m$ sin usar aritmética bignum. Simplemente aplicamos el algoritmo de construcción binaria descrito anteriormente, solo realizando sumas en lugar de multiplicaciones. En otras palabras, hemos .[ex]pandido"la multiplicación de dos números a $O(\log m)$ operaciones de suma y multiplicación por dos (que, en esencia, es una suma).

$$a \cdot b = \begin{cases} 0 & \text{si } a = 0 \\ 2 \cdot \frac{a}{2} \cdot b & \text{si } a > 0 \text{ y } a \text{ par} \\ 2 \cdot \frac{a-1}{2} \cdot b + b & \text{si } a > 0 \text{ y } a \text{ impar} \end{cases}$$

## 7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta técnica:

- DMOJ - Last Digit of $A^B$.
- SPOJ LASTDIG - The last digit
- UVA - 374 - Big Mod
- UVA 11029 - Leading and Trailing