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

Levantamiento a a la potencia de n se expresa ingenuamente como multiplicación por a hecho $n-1$ veces: $a^n = a \cdot a \cdot \ldots \cdot a$. Sin embargo, este enfoque no es práctico para grandes a o n.

$$a^{2b} = a^b \cdot a^b = (a^b)^2$$

La idea de la exponenciación binaria es que dividimos el trabajo usando la representación binaria del exponente.

Vamos a escribir n en base 2, por ejemplo:

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

Desde el número n tiene exactamente $\lfloor \log_2 n \rfloor + 1$ dígitos en base 2, solo necesitamos realizar $O(\log n)$ multiplicaciones, si conocemos las potencias $a^1, a^2, a^4, a^8, \ldots, a^{2^{\lfloor \log n \rfloor}}$.

Entonces sólo necesitamos conocer una forma rápida de calcularlos. Por suerte esto es muy fácil, ya que un elemento de la secuencia es sólo el cuadrado del elemento anterior.

$$3^1 = 3 \tag{1}$$
$$3^2 = \left(3^1\right)^2 = 3^2 = 9 \tag{2}$$
$$3^4 = \left(3^2\right)^2 = 9^2 = 81 \tag{3}$$
$$3^8 = \left(3^4\right)^2 = 81^2 = 6561 \tag{4}$$

Entonces, para obtener la respuesta final para 3^{13}, sólo necesitamos multiplicar tres de ellos (omitiendo 3^2 porque el bit correspondiente en n no está configurado): $3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$

4. Implementación

4.1. C++

Primero, el enfoque recursivo, que es una traducción directa de la fórmula 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;
    }
}
```

El segundo enfoque logra la misma tarea sin recursividad. Calcula todas las potencias en un bucle y multiplica las que tienen el bit establecido correspondiente en n. Aunque la complejidad de ambos enfoques es idéntica, este enfoque será más rápido en la práctica ya que no tenemos la sobrecarga de las llamadas recursivas.

```
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

Primero, el enfoque recursivo, que es una traducción directa de la fórmula 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;
    }
}
```

El segundo enfoque logra la misma tarea sin recursividad. Calcula todas las potencias en un bucle y multiplica las que tienen el bit establecido correspondiente en n. Aunque la complejidad de ambos enfoques es idéntica, este enfoque será más rápido en la práctica ya que no tenemos la sobrecarga de las llamadas recursivas.

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

```
    }
    return res;
}
```

5. 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.

```
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;
}
```

- **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 2×2 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 $O(n \log k)$.

```
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;
}
```

- **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"). Veamos cómo los diferentes tipos de transformaciones cambian las coordenadas:

  - Operación de desplazamiento: añade una constante diferente a cada una de las coordenadas.
  - Operación de escalado: multiplica cada una de las coordenadas por una constante diferente.
  - Operación de rotación: la transformación es más complicada (no entraremos en detalles aquí), pero cada una de las nuevas coordenadas aún se puede representar como una combinación lineal de las antiguas.

  Como puedes ver, cada una de las transformaciones se puede representar como una operación lineal sobre las coordenadas. Por tanto, una transformación se puede escribir como 4×4 matriz de la forma:

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

  que, cuando se multiplica por un vector con las antiguas coordenadas y una unidad, da un nuevo vector con las nuevas coordenadas y una unidad:

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

(¿Por qué introducir una cuarta coordenada ficticia, te preguntarás? Esa es la belleza de las coordenadas homogéneas , que encuentran una gran aplicación en los gráficos por computadora. Sin esto, no sería posible implementar operaciones afines como la operación de desplazamiento como una multiplicación de una sola matriz, como requiere que agreguemos una constante a las coordenadas. ¡La transformación afín se convierte en una transformación lineal en la dimensión superior!)

A continuación se muestran algunos ejemplos de cómo se representan las transformaciones en forma matricial:

- Operación de cambio: cambio x coordinar por 5, y coordinar por 7 y z coordinar por 9.

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

- Operación de escalado: escalar el x coordinar por 10 y los otros dos por 5.

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

- Operación de rotación: girar θ grados alrededor del x eje siguiendo la regla de la mano derecha (sentido antihorario)

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

Ahora bien, una vez que cada transformación se describe como una matriz, la secuencia de transformaciones se puede describir como un producto de estas matrices, y un "bucle"de k. Las repeticiones pueden describirse como la matriz elevada a la potencia de k (que se puede calcular usando exponenciación binaria en $O(\log k)$). De esta manera, la matriz que representa todas las transformaciones se puede calcular primero en $O(m \log k)$, y luego se puede aplicar a cada uno de los n puntos en $O(n)$ para una complejidad total de $O(n + m \log k)$.

- **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).

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

6. Complejidad

La complejidad final de este algoritmo es $O(\log n)$: tenemos que calcular $\log n$ poderes de a, y luego tener que hacer como máximo $\log n$ multiplicaciones para obtener la respuesta final de ellas.

El siguiente enfoque recursivo expresa la misma idea:

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{si } n > 0 \text{ y } n \text{ par} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{si } n > 0 \text{ y } n \text{ impar} \end{cases}$$

7. Ejercicios

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

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

- UVA 1230 - MODEX
- Codeforces - Parking Lot
- Codeforces - Decoding Genome
- Codeforces - Neural Network Country
- Codeforces - Magic Gems
- Codeforces - Stairs and Lines
- leetcode - Count good numbers
- Codechef - Chef and Riffles
- LA - 3722 Jewel-eating Monsters
- CSES - Exponentiation
- CSES - Exponentiation II