Manual de preparación para estudiantes preuniversitarios concursantes del ICPC y IOI en la provincia de Matanzas

Matanzas, 21 de marzo de 2024

Autor de temas: Luis Andrés Valido Fajardo

Índice general

Prólogo

El siguiente documento es una versión mejorada de muchos que comenzaron siendo un simple .txt donde tenía la mala costumbre de colocar el nombre del ejercicio, por la temática por la cual lo había resuelto y el jurado donde lo había solucionado. Creo por ahora este ha superado por mucho los anteriores, quizá dentro de algún tiempo lo vea horrendo. Tuve la necesidad de retomarlo por dos cuestiones. La primera, no se imaginan cuanto me pesaba tener que implementar desde cero ciertos algoritmos sabiendo que los había hecho con anterior para otros ejercicios, así que comencé anotar por temática los ejercicio que iba resolviendo, no es lo mismo tener que hacer la rueda desde cero, a decir creo que para este ejercicio puede coger la rueda de otro y solo tengo pintarla o cambiar el tipo de goma. Lo segundo para ayudar muchachos sobre todo de la que se iniciaban en este mundillo.

La estructura del documento esta acorde según mi punto de vista a las principales áreas que debe dominar un concursante de programación. Dentro de cada área abordaré aquellas temáticas o puntos que de cierta manera domino aunque sea lo básico y que al menos tenga un ejemplo de problema en algún jurado que lo solucione usando ese conocimiento. No obstante si alguien cree que faltó algo que debe ser incluido y se siente en condiciones de abordarlo sin problema se puede poner en contacto conmigo (luis.valido@umcc.cu, luis.valido1989@gmail.com) e incluyó su aporte a este documento.

Cada capítulo termina con el análisis de problemas los cuales se enmarca dentro temas tratados en el capítulo. De no existir ninguna especificación se puede asumir que la solución fue hecha en C++ y el veredicto de ese análisis fue aceptado en ese momento. Bueno creo que por ahora no hay nada más que aclarar. Creo que la cantidad de puntos que pierdo por una ortografía no estandarizada es aceptable por la cantidad de puntos que pueden otorgar los conocimientos aquí expuestos (No obstante se aceptan correciones).Importante casi lo olvidaba este documento digamos que es un beta que se va ir llenando a medida de las posibilidades de tiempo de los autores así que puede encontrar temáticas con solamente el nombre, eso significa que ya se pensado abordarlo lo que aún no se ha tenido el tiempo.

Un poco de historia

Luis Alejandro Arteaga Morales

10^{mo} Bronce en la Copa Regional IPVCE.
11^{no} Medalla de Bronce en el concurso nacional de informática
12^{ce} Preselección nacional de informática
Participó en la Olimpiada Iberoamericana de Informática
Mención en la Olimpiada Iberoamericana de Informática
Bronce en el ICPC entre los equipos preuniversitario

Marcos Cepero

10^{mo}
11^{no}
12^{ce}

Leyán Robaina Mesa

10^{mo}
11^{no}
12^{ce}

Ahmed Rodríguez Martínez

10^{mo}
11^{no}
12^{ce}

José Alejandro Albanés Febles

10^{mo}
11^{no}
12^{ce}

Alejandra Tudela Lara

10^{mo}
11^{no}
12^{ce}

Daniel Alejandro Estupiñan Reyes

10^{mo}
11^{no}
12^{ce}

Adrian Pacheco Rubio

10^{mo}
11^{no}
12^{ce}

Karen Negrín Mazaría

10^{mo}
11^{no}
12^{ce}

Reynier García Montes de Oca

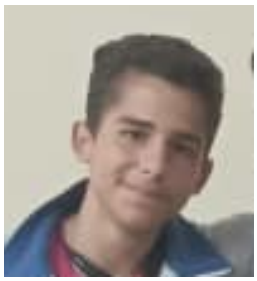
10^{mo}
11^{no}
12^{ce}

Cristian Ernesto Lugo

10^{mo}
11^{no}
12^{ce}

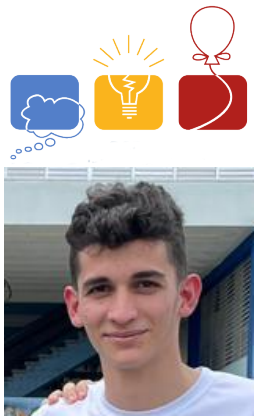
César Raúl Luis González

10^{mo}
11^{no}
12^{ce}



Oscar Manuel Reyes Mora

10^{mo}
11^{no}
12^{ce}



Ronnie Molina Martínez

10^{mo}
11^{no}
12^{ce}

Zaniel García Orihuela

9^{no}
10^{mo}
11^{no}
12^{ce}

ARITMÉTICA ÁLGEBRA

Exponenciación binaria

1.1. 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 ?.

1.2. Conocimientos previos

1.2.1. 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
```

1.2.2. Desplazamiento a izquierda $\ll$

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:

$expr - desplazada \ll expr - desplazamiento$

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.*/
```

1.3. 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.

$$3^1 = 3 \tag{1.1}$$
$$3^2 = \left(3^1\right)^2 = 3^2 = 9 \tag{1.2}$$
$$3^4 = \left(3^2\right)^2 = 9^2 = 81 \tag{1.3}$$
$$3^8 = \left(3^4\right)^2 = 81^2 = 6561 \tag{1.4}$$

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): $3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$

1.4. Implementación

1.4.1. 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;
}
```

1.4.2. 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;
}
```

1.5. 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 mód 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).

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

1.6. 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:

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

1.7. 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

Sucesión de Fibonacci

2.1. Introducción

La conocida sucesión de Fibonacci es aquella sucesión que comienza con los términos 0 y 1, y continua obteniendo cada término sumando sus dos anteriores. Mas formalmente:

$$Fib(n) = \begin{cases} 0 & \text{si} \quad n = 0 \\ 1 & \text{si} \quad n = 1 \\ Fib(n-1) + Fib(n-2) & \text{si} \quad n > 1 \end{cases}$$

Esta sucesión o alguna se sus variantes esta presente en varios problemas de concursos por lo cual es vital que un concursante sepa cuales son las maneras de calcular cualquier elemento de sucesión conociendo el orden de este dentro de la sucesión, así como las propiedades que presenta la sucesión.

2.2. Conocimientos previos

2.2.1. Sucesión númerica

Una **sucesión númerica** es un conjunto ordenado de números, que se llaman **términos** de la sucesión. Cada término se representa por una letra y un subíndice que indica el lugar que ocupa dentro de ella. De acuerdo a la cantidad de términos la msma se puede clasificar como **sucesión finita** cuando la cantidad de términos es finito mientras cuando ocurre lo contrario se dice que estamos en presencia de una **sucesión infinita**. El **término general** ó **término n-ésimo**, a_n, de una sucesión es una fórmula que nos permite calcular cualquier término de la sucesión en función del lugar que ocupa.

2.2.2. Exponenciación Binaria

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).

2.2.3. Multiplicación de matrices

En matemática, la multiplicación o producto de matrices es la operación de composición efectuada entre dos matrices, o bien la multiplicación entre una matriz y un escalar según

unas determinadas reglas.

Vamos a considerar dos matrices:

- La matriz A con n filas y k columnas.
- La matriz B con k filas y m columnas.

Para poder efectuar una multiplicacion entre dos matrices la cantidad de columnas de una debe coincidir con la cantidad de filas de la otra matriz. Si se cumple eso la operación se puede definir como $C = A * B$

$$\begin{bmatrix} a_{11} & a_{12} & ... & a_{1k} \\ a_{21} & a_{22} & ... & a_{2k} \\ ... & ... & ... & ... \\ a_{n1} & ... & ... & a_{nk} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & ... & b_{1m} \\ b_{21} & b_{22} & ... & b_{2m} \\ ... & ... & ... & ... \\ b_{k1} & ... & ... & b_{km} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & ... & c_{1m} \\ c_{21} & c_{22} & ... & c_{2m} \\ ... & ... & ... & ... \\ c_{n1} & ... & ... & c_{nm} \end{bmatrix}$$

Tal que C es una matriz con n filas y m columnas y cada elemento de C se puede calcular con la siguiente fórmula:

$$C_{ij} = \sum_{r=1}^{k} A_{ir} * B_{rj}.$$

2.3. Desarrollo

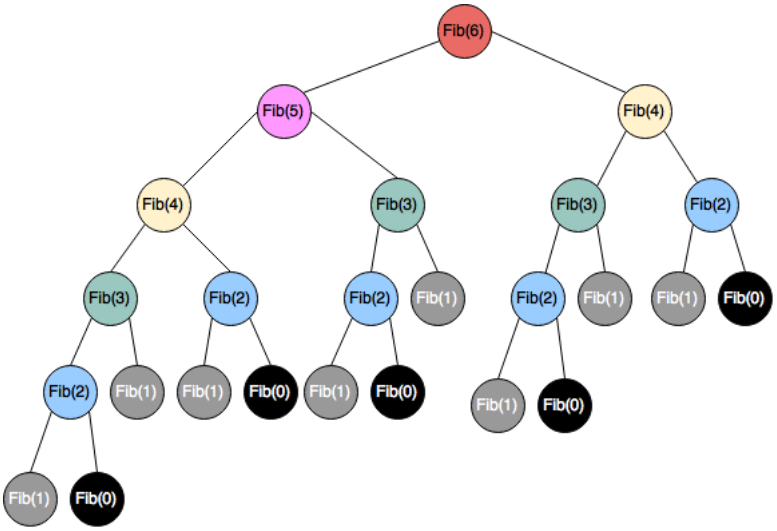
En este orden de ideas, es fácil descubrir los primeros términos de la sucesión:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Del mismo modo podemos expresar cada término en base a su función:
Fib(0)=0, Fib(1)=1, Fib(2)=1, Fib(3)=2, Fib(4)=3, Fib(5)=5, ...

Variante recursiva

Dada esta definición formal, no es nada difícil crear una primera solución recursiva para obtener un término n de la sucesión de Fibonacci. Pero analicemos a fondo lo que hace esta función. Recursivamente, estará llamando a sus dos anteriores. Cada una de estas llamadas, si es mayor que 1, de nuevo estará llamando a sus dos anteriores. Analicemos el árbol de llamadas para un fibonacci de orden 6:

Pero es eficiente? Volvamos al árbol. Para calcular 1 vez Fib(6), es necesario calcular 1 vez Fib(5), 2 veces Fib(4), 3 veces Fib(3), 5 veces Fib(2), 8 veces Fib(1) y cinco veces Fib(0) (Como el algoritmo considera constante Fib(1), solo tenemos 5 llamadas a Fib(0). Si Fib(1) se calculara con su antecesor, tendríamos 13 llamadas). ¡Estamos repitiendo operaciones! Y lo peor, estas operaciones se repiten justamente siguiendo la secuencia de Fibonacci como se puede ver en los números en negrilla (1,1,2,3,5,8,13).



Variante iterativa

Si eliminamos los cálculos repetidos, fácilmente podemos tener un algoritmo de iterativo. Para hacer esto, utilizaremos un array que guarde los valores calculados de forma que en la posición i estará la suma de los valores presentes en $i-1$ y $i-2$ partitiendo que para i igual 0 y 1 esos serán sus valores respectivamente.

Ahora el algoritmo es mucho mas eficiente, reduciendo enormemente el número de operaciones innecesarias para valores altos de n. Pero surge un nuevo problema. La complejidad espacial es de $O(n)$ también. Hemos sacrificado espacio por tiempo, ahora tenemos un array de n posiciones cuando en realidad solo necesitábamos la posición n y las demás podemos desecharlas. Como el algoritmo únicamente necesita de los dos valores anteriores para obtener el siguiente podemos remplazar el array por un grupo de tres variables. Ahora la complejidad temporal sigue siendo de orden $O(n)$ pero la complejidad espacial se reduce a un orden $O(1)$.

Variante matricial

Volviendo al inicio, sabemos que:
$Fib(n) = Fib(n-1) + Fib(n-2)$
Partiendo de aquí, podemos fácilmente desarrollar un sistema de ecuaciones:

$$0Fib(n-2) + 1Fib(n-1) = 1Fib(n-1)$$

$$1Fib(n-2) + 1Fib(n-1) = 1Fib(n-1)$$

Y por supuesto, lo reescribiremos en notación matricial:
$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f(n-2) \\ f(n-2) \end{bmatrix} = \begin{bmatrix} f(n-1) \\ f(n) \end{bmatrix}$
Verificando estos valores para n = 2
$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} f(1) \\ f(2) \end{bmatrix}$
Conociendo que Fib(0)=0 y Fib(1)=1:

$$\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}1\\ 1\end{bmatrix}$$

Y efectivamente, Fib(2)=1. Ahora bien, generalizando para diferentes valores de n.

$$\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1}\begin{bmatrix}0\\ 1\end{bmatrix} = \begin{bmatrix}f(n-1)\\ f(n)\end{bmatrix}$$

$$\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{n-1} = \begin{bmatrix}f(n-2) & f(n-1)\\ f(n-1) & f(n)\end{bmatrix}$$

Ahora recordemos 3 propiedades de la potenciación:

1. $A^1 = A$
2. $A^{m^n} = A^{mn}$
3. $A^n = A^i A^j$ si $i + j = n$

Los cuales nos permitirán hacer un "divide y vencerás" para resolver la potencia de una manera eficiente.

$$A^n = \begin{cases} A & si\ n = 1 \\ (A^{\frac{n}{2}})^2 & si\ n\%2 == 0 \\ A * A^{n-1} & si\ n\%2! = 0 \end{cases}$$

¿Que demonios he hecho? Básicamente, partir el problema. Para explicarlo, miremos paso a paso el proceso para hallar el Fib(11):

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{10}$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{5}\right)^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{4}\right)^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}^{2}\right)^2\right)^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}^{2}\right)^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \left(\begin{bmatrix}0 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}2 & 3\\ 3 & 5\end{bmatrix}\right)^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \begin{bmatrix}3 & 5\\ 5 & 8\end{bmatrix}^2$$

$$\begin{bmatrix}f(9) & f(10)\\ f(10) & f(11)\end{bmatrix} = \begin{bmatrix}34 & 55\\ 55 & 89\end{bmatrix}$$

Y así llegamos a la respuesta de manera mas rápida.

Formula

Esto no es una última mejora, en realidad es un cambio total. Hasta ahora hemos manejado cada término de la sucesión en base a sus términos anteriores. Pero existe también una formula explicita creada por Lucas para hallar directamente un término sin necesidad de iterar.

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Esta fórmula es fácil de probar por inducción, pero se puede deducir con la ayuda del concepto de funciones generadoras o resolviendo una ecuación funcional.

Puedes notar inmediatamente que el valor absoluto del segundo término siempre es menor que 1, y también disminuye muy rápidamente (exponencialmente). Por tanto, el valor del primer término por sí solo es casi F_n. Esto se puede escribir estrictamente como:

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right]$$

donde los corchetes indican redondeo al número entero más cercano.

Como estas dos fórmulas requerirían una precisión muy alta cuando se trabaja con números fraccionarios, son de poca utilidad en cálculos prácticos.

Propiedades de la sucesión de Fibonacci

1. Tan sólo un término de cada tres es par, uno de cada cuatro es múltiplo de 3, uno de cada cinco es múltiplo de 5, etc. Esto se puede generalizar, de forma que la sucesión de Fibonacci es periódica en las congruencias módulo m, para cualquier m.

2. Cualquier número natural se puede escribir mediante la suma de un número limitado de términos de la sucesión de Fibonacci, cada uno de ellos distinto a los demás. Por ejemplo, 17=13+3+1, 65=55+8+2.

3. Cada número de Fibonacci es el promedio del término que se encuentra dos posiciones antes y el término que se encuentra una posición después. Es decir:

   $F_n = \frac{F_{n-2}+F_{n+1}}{2}$

4. Lo anterior también puede expresarse así: calcular el siguiente número a uno dado es 2 veces éste número menos el número 2 posiciones más atrás.

   $F_{n+1} = F_n * 2F_{n-2}$

5. El último dígito de cada número se repite periódicamente cada 60 números. Los dos últimos, cada 300; a partir de ahí, se repiten cada $15 \times 10^{n-1}$ números.

6. La suma de los n primeros números es igual al número que ocupa la posición $n + 2$ menos uno. Es decir

   $F_{n+2} - 1 = F_n ... + F_2 + F_1$

7. La suma de diez números Fibonacci consecutivos es siempre 11 veces superior al séptimo número de la serie.

8. El máximo común divisor de dos números de Fibonacci es otro número de Fibonacci. Más específicamente

   $mcd(F_n, F_m) = F_{mcd(n,m)}$

9. Otras identidades interesantes incluyen las siguientes:

   $f_0 - f_1 + f_2 - \cdots + (-1)^n f_n = (-1)^n f_{n-1} - 1$

   $f_1 + f_3 + f_5 + \cdots + f_{2n-1} = f_{2n}$

   $f_0 + f_2 + f_4 + \cdots + f_{2n} = f_{2n+1} - 1$

   $f_0^2 + f_1^2 + f_2^2 + \cdots + f_n^2 = f_n f_{n+1}$

   $f_1 f_2 + f_2 f_3 + f_3 f_4 + \cdots + f_{2n-1} f_{2n} = f_{2n}^2$

   $f_{n+2}^2 - f_n^2 = f_{2n+2}$

   $f_{n+2}^3 + f_{n+1}^3 - f_n^3 = f_{3n+3}$

10. La identidad de Cassini: $F_{n-1}F_{n+1} - F_n^2 = (-1)^n$

11. La regla de la suma: $F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$

12. Aplicando la identidad anterior al caso $k = n$, obtenemos: $F_{2n} = F_n(F_{n+1} + F_{n-1})$

13. De esto podemos demostrar por inducción que para cualquier número entero positivo k, F_{nk} es múltiplo de F_n.

14. Lo inverso también es cierto: si F_m es múltiplo de F_n, entonces m es múltiplo de n.

15. Los números de Fibonacci son las peores entradas posibles para el algoritmo euclidiano(Teorema de Lame)

Codificación Fibonacci

Podemos usar la secuencia para codificar números enteros positivos en palabras de código binario. Según el teorema de Zeckendorf, cualquier número natural n se puede representar de forma única como una suma de números de Fibonacci:

$$N = F_{k_1} + F_{k_2} + \ldots + F_{k_r}$$

tal que $k_1 \geq k_2 + 2,\ k_2 \geq k_3 + 2,\ \ldots,\ k_r \geq 2$ (es decir: la representación no puede utilizar dos números de Fibonacci consecutivos).

De ello se deduce que cualquier número puede codificarse de forma única en la codificación de Fibonacci. Y podemos describir esta representación con códigos binarios $d_0 d_1 d_2 \ldots d_s 1$, dónde d_i es 1 si F_{i+2} se utiliza en la representación. El código irá acompañado de un 1 para indicar el final de la palabra clave. Observe que este es el único caso en el que aparecen dos bits 1 consecutivos.

$$\begin{array}{rcllcl} 1 & = & 1 & = & F_2 & = & (11)_F \\ 2 & = & 2 & = & F_3 & = & (011)_F \\ 6 & = & 5+1 & = & F_5+F_2 & = & (10011)_F \\ 8 & = & 8 & = & F_6 & = & (000011)_F \\ 9 & = & 8+1 & = & F_6+F_2 & = & (100011)_F \\ 19 & = & 13+5+1 & = & F_7+F_5+F_2 & = & (1001011)_F \end{array}$$

La codificación de un número entero n se puede hacer con un algoritmo codicioso simple:

1. Repita los números de Fibonacci del mayor al menor hasta encontrar uno menor o igual a n.

2. Supongamos que este número fuera F_i. Sustraer F_i de n y poner un 1 en el $i-2$ posición de la palabra de código (indexación desde 0 desde el bit más a la izquierda hasta el más a la derecha).

3. Repetir hasta que no quede resto.

4. Añadir una final 1 a la palabra clave para indicar su final.

Para decodificar una palabra clave, primero elimine la final 1. Entonces, si el i-ésimo bit está configurado (indexado desde 0 desde el bit más a la izquierda hasta el más a la derecha), suma F_{i+2} al número.

Módulo de periodicidad p

Considere el módulo de la secuencia de Fibonacci p. Probaremos que la secuencia es periódica.

Demostremos esto por contradicción. Considere el primero p^2+1 pares de números de Fibonacci tomados módulo p:

$$(F_0,\ F_1),\ (F_1,\ F_2),\ \ldots,\ (F_{p^2},\ F_{p^2+1})$$

Sólo puede haber p módulo de restos diferentes p, y como mucho p^2 diferentes pares de restos, por lo que hay al menos dos pares idénticos entre ellos. Esto es suficiente para demostrar que la secuencia es periódica, ya que un número de Fibonacci sólo está determinado por sus dos predecesores. Por lo tanto, si dos pares de números consecutivos se repiten, eso también significaría que los números posteriores al par se repetirán de la misma manera.

Ahora elegimos dos pares de restos idénticos con los índices más pequeños de la secuencia. Deja que los pares sean $(F_a,\ F_{a+1})$ y $(F_b,\ F_{b+1})$. Lo demostraremos $a=0$. Si esto fuera falso, habría dos pares anteriores $(F_{a-1},\ F_a)$ y $(F_{b-1},\ F_b)$, que, según la propiedad de los números de Fibonacci, también sería igual. Sin embargo, esto contradice el hecho de que habíamos elegido pares con los índices más pequeños, completando nuestra prueba de que no existe un preperíodo (es decir, los números son periódicos a partir de F_0).

2.4. Implementación

2.4.1. C++

Variante recursiva

```
int fibonacci(int n){
   if(n<2) return n;
   else return fibonacci(n-1)+fibonacci(n-2);
}
```

Variante iterativa

```
int fibonacci[1000];

// Complejidad temporal y espacial O(n)
void buildFibonacci(){
   fibonacci[0]=0; fibonacci[1]=1;
   for(int i=2;i<1000;i++) fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
}

// Complejidad temporal y espacial O(n) y O(1) respectivamente
int fibonacci(int n){
   int a=0, b=1,c=n;
   for(int i=2;i<=n;i++){ c=a+b; a=b; b=c; }
   return c;
}
```

Variante matricial

```
int fibonacci(int n){
   int h,i,j,k,aux; h=i=1; j=k=0;
   while(n>0){
      if(n%2!=0){
         aux=h*j; j=h*i+j*k+aux; i=i*k+aux;
      }
      aux=h*h; h=2*h*k+aux; k=k*k+aux; n=n/2;
    }
    return j;
}

//Este metodo devuelve el par Fn y Fn+1
pair<int, int> fib (int n) {
   if (n == 0) return {0, 1};
   auto p = fib(n >> 1);
   int c = p.first * (2 * p.second - p.first);
   int d = p.first * p.first + p.second * p.second;
```

```
    if (n & 1) return {d, c + d};
    else return {c, d};
}

//otra variante
struct matrix {
    long long mat[2][2];
    matrix friend operator *(const matrix &a, const matrix &b){
        matrix c;
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c.mat[i][j] = 0;
                for (int k = 0; k < 2; k++) {
                    c.mat[i][j] += a.mat[i][k] * b.mat[k][j];
                }
            }
         }
         return c;
    }
};

matrix matpow(matrix base, long long n) {
    matrix ans{ {
        {1, 0},
        {0, 1}
    } };
    while (n) {
        if(n&1)
            ans = ans*base;
            base = base*base;
            n >>= 1;
        }
    return ans;
}

long long fib(int n) {
    matrix base{ { {1, 1},{1, 0} } };
    return matpow(base, n).mat[0][1];
}
```

Función

```
double fibonacci(int n){
    double root=sqrt(5);
    double fib=((1/root)*(pow((1+root)/2,n))- (1/root)*(pow((1-root)/2,n)));
    return fib;
}
```

2.4.2. Java

Variante recursiva

```
public int fibonacci(int n){
    if(n<2) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
```

Variante iterativa

```
int [] fibonacci = new int[1000];

// Complejidad temporal y espacial O(n)
public void buildFibonacci(){
    fibonacci[0]=0; fibonacci[1]=1;
    for(int i=2;i<1000;i++) fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];
}

// Complejidad temporal y espacial O(n) y O(1) respectivamente
public int fibonacci(int n){
    int a=0, b=1,c=n;
    for(int i=2;i<n;i++){ c=a+b; a=b; b=c; }
    return c;
}
```

Variante matricial

```
public int fibonacci(int n){
    int h,i,j,k,aux; h=i=1; j=k=0;
    while(n>0){
        if(n%2!=0){
            aux=h*j; j=h*i+j*k+aux; i=i*k+aux;
        }
        aux=h*h; h=2*h*k+aux; k=k*k+aux; n=n/2;
    }
    return j;
}
```

Función

```
public double fibonacci(int n){
    double root=Math.sqrt(5);
    double fib=((1/root)*(Math.pow((1+root)/2,n))- (1/root)*(Math.pow((1-root)
        /2,n)));
    return fib;
```

```
}
```

2.5. Aplicaciones

Dentro de las sucesiones conocidas la de Fibonacci es de las más utilizadas en problemas de concursos siempre con alguna que otra variación o con la aplicación de algunas de las propiedades que ella presente es por ello la importancia de concerla y las formas de hallar determinado término de la sucesión también tiene diversas aplicaciones en diferentes campos, algunas de ellas son:

1. **Matemáticas:** La secuencia de Fibonacci es utilizada en matemáticas para estudiar patrones de crecimiento y propiedades numéricas.

2. **Computación:** La secuencia de Fibonacci es utilizada en algoritmos y programas informáticos, como por ejemplo en la optimización de códigos y en la programación dinámica.

3. **Biología:** La secuencia de Fibonacci se encuentra en la naturaleza, en patrones de crecimiento de plantas, en la disposición de las hojas en las ramas, en la estructura de las conchas y en la distribución de semillas en los girasoles, entre otros.

4. **Finanzas:** La secuencia de Fibonacci es utilizada en el análisis técnico de los mercados financieros para predecir tendencias y movimientos de precios.

5. **Arte y diseño:** La secuencia de Fibonacci es utilizada en arte y diseño para crear composiciones equilibradas y armoniosas, ya que se considera que los números de Fibonacci representan proporciones estéticamente agradables.

2.6. Complejidad

La solución recursiva es un algoritmo de orden exponencial, mas exactamente ϕ^n. Esto significa que es terriblemente lento, pues tendrá que hacer excesivas operaciones para valores altos de n.

La solución iterativa tiene una complejidad tanto espacial como temporal de $O(n)$ siendo n el término enésimo de fibonacci a calcular.

La solución matricial su complejidad es $O(log_2 n)$. Para hacernos una idea, para calcular el Fib(200) con el algoritmo exponencial, ocuparíamos $6{,}2737x10^{41}$ cálculos (interminable). Con el algoritmo de orden n ocuparíamos 200 cálculos, y con este algoritmo, 8 operaciones (Por supuesto estos datos son aproximados, pero muestran claramente las diferencias abismales).

En cuanto a la función es muy eficiente, pero su orden depende de la manera de implementar la potenciación. Una potenciación por cuadrados como en el caso anterior, arroja un costo de $O(log_2 n)$. Pero se debe tener mucho cuidado en su implementación, pues cada lenguaje tiene diferentes maneras de implementar las funciones matemáticas, y muy posiblemente sea necesario redondear el número para dar la respuesta exacta.

2.7. Ejercicios

El siguiente listado de problemas para su solución debemos apoyarnos en la suseción de Fibonacci.

- DMOJ - Salto de rana
- DMOJ - Fibonacci 2D
- DMOJ - Secuencia numerada de lapices
- DMOJ - Fibonacci Calculation
- SPOJ - Euclid Algorithm Revisited
- SPOJ - Fibonacci Sum
- Codeforces - C. Fibonacci
- Codeforces - A. Hexadecimal's theorem
- Codeforces - B. Blackboard Fibonacci
- Codeforces - E. Fibonacci Number
- LightOJ - Number Sequence
- HackerRank - Is Fibo
- Project Euler - Even Fibonacci numbers
- DMOJ - Fibonacci Sequence
- DMOJ - Fibonacci Sequence (Harder)

ESTRUCTURA DE DATOS

Estructuras de datos basadas en políticas (árbol de estadísticas de pedidos)

3.1. Introducción

Un tipo de estructura de datos que se utiliza comúnmente en informática es la estructura de datos basada en políticas. Una estructura de datos basada en políticas es una estructura de datos que permite al programador definir un conjunto de reglas o políticas que gobiernan el comportamiento de la estructura de datos.

Por ejemplo, suponga que tiene un conjunto de datos que desea almacenar en una estructura de datos. Podría utilizar una estructura de datos estándar como una matriz o una lista vinculada para almacenar los datos. Sin embargo, si desea aplicar ciertas políticas sobre los datos, como permitir que solo se almacenen ciertos valores o garantizar que los datos siempre estén ordenados en un orden particular, es posible que necesite utilizar una estructura de datos basada en políticas.

Las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, son estructuras de datos especializadas que permiten consultar y actualizar datos de manera eficiente en función de políticas o reglas específicas.

3.2. Conocimientos previos

3.2.1. Árbol de búsqueda binario equilibrado

Un árbol de búsqueda binario equilibrado, también conocido como árbol AVL (por las iniciales de los inventores Adelson-Velsky y Landis), es una estructura de datos en forma de árbol binario de búsqueda en la que se garantiza que la diferencia de alturas entre los subárboles izquierdo y derecho de cada nodo no sea mayor que 1. Esto asegura que el árbol esté equilibrado y mantiene su eficiencia en términos de tiempo de búsqueda, inserción y eliminación.

3.2.2. Árboles rojo-negro

Un árbol rojo-negro es otra estructura de datos en forma de árbol binario de búsqueda que se utiliza para mantener el equilibrio y garantizar un rendimiento eficiente en términos de tiempo de búsqueda, inserción y eliminación. Los árboles rojo-negro se caracterizan por

tener nodos que son rojos o negros, y cumplen con ciertas reglas para mantener el equilibrio del árbol.

Las reglas fundamentales de un árbol rojo-negro son las siguientes:

1. Cada nodo es rojo o negro.
2. La raíz del árbol siempre es negra.
3. Todos los nodos hoja (nodos nulos) son negros.
4. Si un nodo es rojo, entonces sus hijos deben ser negros.
5. Para cualquier nodo dado, todos los caminos desde ese nodo hasta sus nodos hoja descendientes contienen el mismo número de nodos negros.

Estas reglas garantizan que la altura negra de cualquier camino desde la raíz hasta un nodo hoja sea la misma, lo que mantiene el equilibrio del árbol y asegura un rendimiento eficiente en las operaciones de búsqueda, inserción y eliminación.

Durante las operaciones de inserción y eliminación en un árbol rojo-negro, se aplican rotaciones y cambios de colores en los nodos para mantener las propiedades del árbol. Estas operaciones se realizan de manera similar a los árboles AVL, pero con reglas específicas para los colores de los nodos.

3.2.3. PBDS

PBDS es una abreviatura que puede referirse a *Policy-Based Data Structures* en el contexto de la biblioteca de estructuras de datos estándar de C++ (STL). Las PBDS son una extensión de la STL que proporciona una forma flexible de definir estructuras de datos con políticas personalizadas para operaciones como inserción, eliminación, búsqueda, etc. Esto permite a los desarrolladores adaptar las estructuras de datos a sus necesidades específicas mediante la especificación de políticas en lugar de implementar estructuras de datos personalizadas desde cero.

En el caso de los árboles rojo-negro, la biblioteca PBDS en C++ ofrece una implementación eficiente y flexible de árboles rojo-negro que se pueden personalizar con políticas específicas según los requisitos del usuario. Esta característica hace que sea más fácil y conveniente trabajar con estructuras de datos complejas como los árboles rojo-negro en C++, ya que se pueden adaptar a diferentes escenarios y requisitos sin tener que reescribir toda la estructura desde cero.

3.3. Desarrollo

Las estructuras de datos basadas en políticas en C++ son algo similares a los conjuntos. Proporcionan algunas operaciones adicionales, pero considerablemente poderosas, que otorgan al programador las virtudes de alto rendimiento, seguridad semántica y flexibilidad en comparación con las estructuras de datos estándar de la biblioteca estándar de C++.

Un árbol de estadísticas de orden es un tipo de árbol de búsqueda binario equilibrado que mantiene información adicional sobre el orden de los elementos del árbol. Esta información

permite la recuperación rápida del k-ésimo elemento más pequeño del árbol, así como otras consultas relacionadas con pedidos.

La implementación de un árbol de estadísticas de pedidos normalmente implica aumentar un árbol de búsqueda binario estándar con campos u operaciones adicionales para mantener la información de las estadísticas de pedidos. Esto se puede hacer utilizando técnicas como árboles rojo-negro o árboles AVL.

3.3.1. Conjunto ordenado

Los conjuntos ordenados pueden verse como versiones extendidas de conjuntos en la estructura de datos basada en políticas.

3.3.2. Operaciones

Hay dos operaciones adicionales en las estructuras de datos basadas en políticas, además de las de la biblioteca estándar. Se enumeran a continuación:

- **order_of_key(k):** Devuelve el número de elementos estrictamente menores que k.
- **find_by_order(k):** Devuelve la dirección del elemento en el késimo índice del conjunto mientras se utiliza la indexación basada en cero, es decir, el primer elemento está en el índice cero.

Aclarando las operaciones adicionales que ofrece PBDS:
Sea a un conjunto ordenado en el que se insertan los elementos 2, 4, 3, 7, 5.
Entonces los elementos en un conjunto ordenado a se almacenarían en el siguiente orden:

$$2,\ 3,\ 4,\ 5,\ 7$$

- **find_by_order:** Aquí, *find_by_order(x)* devolverá la dirección del (x+1)-ésimo elemento en el conjunto ordenado a. Por ejemplo, *find_by_order(3)* devolverá la dirección de $'5'$ ya que el cuarto elemento (usando indexación basada en 1) presente en el conjunto es 5.
- **order_of_key:** Aquí, *order_of_key(x)* devolverá el número de elementos estrictamente menor que x. Por ejemplo, *order_of_key(10)* devolverá 5, ya que todas las entradas del conjunto son menores de 10.

Supongamos que tenemos una situación donde los elementos se insertan uno por uno en un arreglo y después de cada inserción, se nos da un rango $[l, r]$ y tenemos que determinar el número de elementos en el arreglo mayor que igual a l y menor igual a r. Inicialmente, el arreglo está vacío.

Aplicando esta estructura con estas funciones podemos resolver el problema anterior fácilmente, es decir, el recuento de elementos entre l y r se puede encontrar mediante:

```
o_set.order_of_key(r+1) - o_set.order_of_key(l)
```

Como el conjunto contiene solo los elementos ÚNICOS, para realizar las operaciones en un arreglo que tiene elementos repetidos, podemos tomar la CLAVE como un par de elementos en lugar de un número entero en el que el primer elemento es nuestro elemento requerido del arreglo y solo el segundo elemento. del par debe ser único para que todo el par sea único.

3.4. Implementación

Esta estructura solo está presente en el lenguaje de programación C++.

```
//Incluya los siguientes archivos de encabezado en su codigo para usar PBDS:
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
//Namespace
using namespace __gnu_pbds;
//Plantillas
//definir plantilla cuando todos los elementos son distintos
template <class T> using nombre_tu_estructura_v1 = tree<T, null_type,
less<T>, rb_tree_tag,tree_order_statistics_node_update>;
//definir plantilla cuando tambien se utilizan elementos duplicados
template <class T> using nombre_tu_estructura_v2 = tree<T, null_type,
less_equal<T>, rb_tree_tag,tree_order_statistics_node_update>;

nombre_tu_estructura_v1 <int> B;
nombre_tu_estructura_v1 <pair<int,int> > A;

nombre_tu_estructura_v2 <int> b;
nombre_tu_estructura_v2 <pair<int,int> > a;
```

- **T:** T hace referencia al tipo de dato que va almacenar la estructura.
- **null_type:** Es la política mapeada. Aquí es nulo usarlo como un conjunto. Si queremos obtener el mapa pero no el conjunto, como segundo argumento se debe usar el tipo mapeado.
- **less:** Es la base para la comparación de dos funciones.
- **rb_tree_tag:** tipo de árbol utilizado. Generalmente son árboles rojos y negros porque la inserción y eliminación tardan un tiempo $\log(N)$, mientras que otros tardan un tiempo lineal, como *splay_tree*. Hay tres clases base proporcionadas en STL para esto: *rb_tree_tag* (árbol rojo-negro), *splay_tree_tag* (árbol de visualización) y *ov_tree_tag* (árbol de vectores ordenados). Lamentablemente, en las competiciones solo podemos usar árboles rojo-negro para esto porque el árbol splay y el árbol OV usan una operación de división cronometrada lineal que nos impide usarlos.
- **tree_order_statistics_node__update:** Está incluido en *tree_policy.hpp* y contiene varias operaciones para actualizar las variantes de nodos de un contenedor basado en árbol, de modo que podamos realizar un seguimiento de metadatos como el número de nodos en un subárbol. Por defecto está configurado en *null_node_update*, es decir,

información adicional no almacenada en los vértices. Además, C++ implementó una política de actualización *tree_order_statistics_node_update*, que, de hecho, lleva a cabo las operaciones necesarias.

```
#include <iostream>
#include <bits/stdc++.h>
//Biblioteca para trabajar con la estructura
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
using namespace std;
//namespace para trabajar con la estructura
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update>
ordered_set;

int main(){
    ordered_set p;
    p.insert(5); p.insert(2); p.insert(6); p.insert(4);
    // valor en el tercer indice de la matriz ordenada.
    cout << "El valor en el tercer indice::"<< *p.find_by_order(3) << endl;
    // indice del numero 6
    cout << "El indice del numero 6::" << p.order_of_key(6)<< endl;
    // El numero 7 no esta en el conjunto, pero mostrara el
    // numero de indice si estaba alli en la matriz ordenada.
    cout << "El indice del numero siete ::"<< p.order_of_key(7) << endl;
    return 0;
}
```

Para insertar múltiples copias del mismo elemento en el conjunto ordenado, un enfoque simple es reemplazar la siguiente línea,

```
typedef tree<int , null_type, less<int> , rb_tree_tag ,
    tree_order_statistics_node_update> ordered_set;
```

Por

```
typedef tree<int , null_type , less_equal<int> , rb_tree_tag ,
    tree_order_statistics_node_update> ordered_multiset
```

Como se muestra en el siguiente ejemplo:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
tree_order_statistics_node_update>
```

```
ordered_multiset;


int main(){
    ordered_multiset p;
    p.insert(5); p.insert(5);
    p.insert(5); p.insert(2);
    p.insert(2); p.insert(6);
    p.insert(4);
    for (int i = 0; i < (int)p.size(); i++) {
        cout << "El elemento presente en el indice " << i << " es ";
        // Elemento de impresion presente en el indice i
        cout << *p.find_by_order(i) << ' ';
        cout << '\n';
    }
    return 0;
}
```

Sin embargo, el enfoque anterior no se recomienda porque la estructura de datos basados en políticas de G++ no está diseñada para almacenar elementos en un orden estricto y débil, por lo que usar *less_equal* con ella puede provocar un comportamiento indefinido, como que *std::find* proporcione resultados incorrectos y búsquedas más largas. veces. Otro enfoque es almacenar los elementos como pares donde el primer elemento es el valor del elemento y el segundo elemento del par es el índice en el que se encontró en la matriz. El código fuente a continuación demuestra cómo utilizar este enfoque junto con otros métodos disponibles en el conjunto ordenado.

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <functional> // for less
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

typedef tree<pair<int, int>, null_type,
less<pair<int, int> >, rb_tree_tag,
tree_order_statistics_node_update>
ordered_multiset;

int main(){
    ordered_multiset p;

    // Durante la insercion, utilice {VAL, IDX}
    p.insert({ 5, 0 }); p.insert({ 5, 1 });
    p.insert({ 5, 2 }); p.insert({ 2, 3 });
    p.insert({ 2, 4 }); p.insert({ 6, 5 });
    p.insert({ 4, 6 });

    for (int i = 0; i < (int)p.size(); i++) {
        cout << "El elemento presente en el indice " << i << " es ";
        // Elemento de impresion presente en el indice i
        // Utilice .first para obtener el valor real
        cout << p.find_by_order(i)->first << ' ';
```

```
        cout << '\n';
    }
    cout << "\nDeleting element of 2\n\n";
    // Siempre que busque, utilice {VAL, 0} para lower_bound
    //                             {VAL, IDX} para buscar
    //                             {VAL, INT_MAX} para upper_bound
    p.erase(p.lower_bound({ 2, 0 }));
    for (int i = 0; i < (int)p.size(); i++) {
        cout << "El elemento presente en el indice " << i << " es ";
        cout << p.find_by_order(i)->first << ' ';
        cout << '\n';
    }
    return 0;
}
```

3.5. Aplicaciones

Uno de los beneficios de utilizar una estructura de datos basada en políticas es que le permite separar el almacenamiento de datos de la manipulación de datos. Esto puede hacer que su código sea más modular y más fácil de mantener.

El árbol de estadísticas de orden es útil en escenarios en los que necesita encontrar rápidamente el k-ésimo elemento más pequeño en un conjunto de elementos o mantener un orden de elementos con inserciones y eliminaciones eficientes.

Las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, tienen diversas aplicaciones en diferentes áreas, incluyendo:

1. **Bases de datos:** En bases de datos relacionales, las *Policy-based Data Structures* pueden ser utilizadas para optimizar consultas que requieren recuperar elementos en un orden específico, como el k-ésimo elemento más pequeño o más grande.

2. **Algoritmos de búsqueda:** En algoritmos de búsqueda y ordenamiento, el *Order Statistics Tree* puede ser utilizado para encontrar rápidamente elementos en un orden específico, lo que es útil en algoritmos de búsqueda binaria y otros algoritmos de búsqueda eficientes.

3. **Estadísticas y análisis de datos:** En aplicaciones que requieren cálculos estadísticos o análisis de grandes conjuntos de datos, las estructuras basadas en políticas pueden ser utilizadas para mantener información sobre el orden de los datos y facilitar la recuperación de estadísticas específicas, como la mediana o percentiles.

4. **Sistemas de información geográfica (GIS):** En sistemas de información geográfica, las *Policy-based Data Structures* pueden ser empleadas para organizar y consultar datos espaciales en un orden específico, por ejemplo, para encontrar rápidamente los puntos más cercanos a una ubicación dada.

5. **Juegos y simulaciones:** En aplicaciones de juegos y simulaciones, las estructuras de datos basadas en políticas pueden ser utilizadas para gestionar y acceder a elementos

en un orden específico, como los jugadores con la puntuación más alta o los eventos en un juego en un orden cronológico.

En resumen, las estructuras de datos basadas en políticas, como el árbol de estadísticas de pedidos, son herramientas versátiles que pueden ser aplicadas en una amplia variedad de contextos donde se requiere un acceso eficiente ya sea en forma de consulta o actualización de los datos en función de políticas o reglas específicas.

3.6. Complejidad

Al ser una estructura implementadas basada en conjunto (*set*) sus operaciones tienen una complejidad logarítmica y lo mismo pasa con las dos operaciones analizadas de esta estructura.

3.7. Ejercicios

A continuación una lista de ejercicios ue se pueden resolver aplicando esta estructura de datos:

- CSES - Josephus Problem II
- Timus Online Judge - 1028. Stars
- Timus Online Judge - 1090. In the Army Now
- Timus Online Judge - 1521. War Games 2
- Timus Online Judge - 1439. Battle with You-Know-Who
- Kattis - Galactic Collegiate Programming Contest