Propiedades matemáticas interesantes

Jaime Sebastian Chavarria Fuertes

December 25, 2024

1 Fibonacci

1.1 Calculo eficiente del n-simo fibonacci en $O(\log n)$

Los números de Fibonacci están definidos por la relación:

$$F(n) = F(n-1) + F(n-2), \quad \text{con } F(0) = 0, F(1) = 1.$$

Esta relación puede expresarse matricialmente como:

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}.$$

La matriz de transición es:

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

Generalizacion mediante potencias:

Iterando este proceso, se obtiene:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = T^{n-1} \cdot \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}.$$

Por lo tanto, $F(n)$ es el elemento en la esquina superior izquierda de T^{n-1}:

$$F(n) = T^{n-1}[0][0].$$

La implementacion en C++ se encuentra en el notebook en la pagina XX

2 Problemas Ad Hoc

2.1 Aplicar una permutación k veces

Este problema trata sobre aplicar una **permutación** varias veces a una secuencia.

2.1.1 ¿Qué es una permutación?

Una permutación en este contexto es una reorganización de los índices de una secuencia. Por ejemplo, si tienes una secuencia:

$$\text{sequence} = [10, 20, 30]$$

$$\text{permutation} = [2, 0, 1]$$

Esto significa que:

- El elemento en la posición 0 va a la posición 2 (el 10 se mueve al índice 2).
- El elemento en la posición 1 va a la posición 0 (el 20 se mueve al índice 0).
- El elemento en la posición 2 va a la posición 1 (el 30 se mueve al índice 1).

El resultado de aplicar esta permutación una vez sería:

$$[20, 30, 10]$$

2.1.2 Objetivo del problema

Queremos aplicar esta permutación no solo una vez, sino k **veces**, a una secuencia inicial.

2.1.3 Solución inicial: Aplicación directa

Podríamos simplemente aplicar la permutación al vector k veces con un bucle:

```
for (int i = 0; i < k; i++) {
    sequence = applyPermutation(sequence, permutation);
}
```

Sin embargo, esto tendría una complejidad de $O(n \cdot k)$, donde n es el tamaño de la secuencia. Esto no es eficiente si k es grande.

2.1.4 Optimización con exponenciación binaria

Concepto clave: Exponenciación binaria

La idea es que podemos aplicar la permutación de forma más eficiente aprovechando las potencias de 2:

1. Si queremos aplicar la permutación $k = 13$ veces, podemos descomponer k en potencias de 2:
   $$13 = 1 + 4 + 8 \quad \text{(o en binario 1101)}$$
   Esto significa:
   - Aplicamos la permutación una vez ($1 = 2^0$).
   - Luego la aplicamos 4 veces ($4 = 2^2$).
   - Luego 8 veces ($8 = 2^3$).
2. En lugar de calcular las permutaciones de forma individual, podemos "acelerarlo":
   - Calcula la permutación P^2 aplicando P sobre sí misma ($P^2 = P \circ P$).
   - Calcula P^4 aplicando P^2 sobre sí misma, y así sucesivamente.
3. Finalmente, suma las permutaciones correspondientes según la representación binaria de k.

2.1.5 Implementación

En el código, se usa `applyPermutation` para aplicar una permutación a la secuencia o a la permutación misma.

- `applyPermutation`: Toma una secuencia y aplica una permutación.
- `permute`: Calcula la secuencia final después de aplicar la permutación k veces utilizando exponenciación binaria.

2.1.6 Segunda solución: Uso de ciclos

Una permutación puede representarse como un **grafo de ciclos**, donde cada índice apunta a su nueva posición. Por ejemplo:

$$\text{permutation} = [2, 0, 1]$$

Esto corresponde al grafo:

$$0 \to 2 \to 1 \to 0 \quad \text{(un ciclo)}$$

2.1.7 Idea clave

1. Encuentra los **ciclos** en el grafo de la permutación.
2. Para cada ciclo, aplica solo $k \mod$ longitud del ciclo. Esto es suficiente porque después de recorrer el ciclo completamente, los elementos vuelven a sus posiciones originales.

2.1.8 Ventaja

Este enfoque tiene una complejidad de $O(n)$, ya que procesamos cada elemento una sola vez mientras encontramos los ciclos.

2.1.9 Implementación

La implementacion esta en la pagina XX del notebook