Universidad Nacional de San Agustín de Arequipa

Facultad de Ingeniería de Producción y Servicios

Escuela Profesional de Ingeniería de Sistemas

INFORME DE TRABAJO GRUPAL

Estructura de datos y Algoritmos

Alumnos:

- Payehuanca Riquelme Jhastyn Jefferson
- Zapata Butron Reyser Julio

Docente:
Quispe Vergaray Karen Melissa

30 de septiembre 2023
Arequipa - Perú

Informe Trabajo Grupal

Tema: Técnicas de Diseño de Algoritmos

1. URL de Repositorio Github

Link del Repositorio en Github: `https://github.com/ReyserLyn/eda-lab2.git`

2. Ejercicios designados

- Fibonacci recursivo
- Fibonacci iterativo
- Fibonacci con memorización no iterativo
- Fibonacci logaritmico
- Informe sobre prueba realizadas para valores n grandes (1000, 10000, 1000000 ...)

3. Solución

Explicación de la función Main

- El programa comienza por procesar los argumentos de la línea de comandos, donde args[0] se interpreta como el número para el cual se calculará el valor de Fibonacci y args[1] sería una indicación si mostrar el numero resultante o no.
- Se registra el tiempo de inicio (tmpInicio) antes de realizar el cálculo.
- Luego, se calcula el valor de Fibonacci llamando a la función fibonacci_iterativo(x).
- Se registra el tiempo de finalización (tmpFin) después de calcular el valor de Fibonacci y se calcula el tiempo transcurrido en milisegundos (tmpTrans).
- Si el segundo argumento (args[1]) proporcionado es "true", el programa imprime el resultado de Fibonacci.
- Finalmente, se imprime el tiempo de ejecución en milisegundos.

3.1. Fibonacci Recursivo

Para la implementación recursiva de la secuencia de Fibonacci se calcula mediante llamadas a funciones recursivas. En este enfoque, el número de Fibonacci en un índice n dado se calcula sumando recursivamente los números de Fibonacci en los índices n-1 y n-2, con casos base definidos para n=0 y n=1.

Listing 1: Fib_Recursivo.java

```java
import java.math.BigInteger;

public class Fib_Recursivo {
    public static void main(String[] args) {
```

```
        int x = (args.length > 0) ? Integer.parseInt(args[0]) : 0;

        long tmpInicio = System.currentTimeMillis();

        BigInteger result = fibonacci_recursivo(x);

        long tmpFin = System.currentTimeMillis();
        long tmpTrans = tmpFin - tmpInicio;

        if (args.length > 1 && args[1].equals("true")) {
            System.out.println("\nFibonacci Recursivo (" + x + ") = " + result);
        }

        System.out.println("\nTiempo de ejecucion: " + tmpTrans + " milisegundos\n\n");
    }

    public static BigInteger fibonacci_recursivo(int n) {
        if (n <= 1) {
            return BigInteger.valueOf(n);
        } else {
            return fibonacci_recursivo(n - 1).add(fibonacci_recursivo(n - 2));
        }
    }
}
```

Explicación de la función fibonacci_recursivo

- If $n \leq 1$, then return BigInteger.valueOf(n): Si n es menor o igual a 1 (es decir, es un caso base), este método devuelve BigInteger.valueOf(n), que es 0 o 1, dependiendo del valor de n.
- Otherwise, return fibonacci_recursivo($n-1$) + fibonacci_recursivo($n-2$): Si n es mayor que 1 (es decir, no es un caso base), el método calcula el número de Fibonacci de forma recursiva llamándose a sí mismo con $n-1$ y $n-2$, y luego sumando los resultados. Este enfoque recursivo continúa hasta llegar a los casos base.

3.2. Fibonacci Iterativo

Para la implementación iterativa de Fibonacci se utliza un bucle para calcular los números de la secuencia en orden ascendente. Es eficiente y evita los problemas de desbordamiento de pila que pueden ocurrir con la implementación recursiva cuando se calculan números grandes. En esta implementación, se utiliza el tipo de dato BigInteger para manejar números grandes que puedan generarse en la secuencia de Fibonacci.

Listing 2: Fib_Iterativo.java

```

import java.math.BigInteger;

public class Fib_Iterativo {
  public static void main(String[] args) {
    int x = (args.length > 0) ? Integer.parseInt(args[0]) : 0;

    long tmpInicio = System.currentTimeMillis();

    BigInteger result = fibonacci_iterativo(x);

```

```
    long tmpFin = System.currentTimeMillis();
    long tmpTrans = tmpFin - tmpInicio;

    if (args.length > 1 && args[1].equals("true")){
      System.out.println("\nFibonacci Iterativo (" + x + ") = " + result);
    }

    System.out.println("\nTiempo de ejecucion: " + tmpTrans + " milisegundos\n\n");
  }

  public static BigInteger fibonacci_iterativo(int num) {
    if (num <= 1) {
      return BigInteger.valueOf(num);
    }

    BigInteger fibPrevPrev = BigInteger.ZERO;
    BigInteger fibPrev = BigInteger.ONE;
    BigInteger fibCurrent = BigInteger.ZERO;

    for (int i = 2; i <= num; i++) {
      fibCurrent = fibPrev.add(fibPrevPrev);
      fibPrevPrev = fibPrev;
      fibPrev = fibCurrent;
    }

    return fibCurrent;
  }
}
```

Explicación de la función fibonacci_iterativo

- Se manejan los casos base para 0 y 1. Si num es igual o menor a 1, se devuelve el valor correspondiente (0 o 1) como un objeto BigInteger.
- Dentro del bucle for, se calcula el valor de Fibonacci iterativamente utilizando tres objetos BigInteger: fibPrevPrev, fibPrev, y fibCurrent.
- Los valores anteriores se actualizan en cada iteración para calcular el siguiente valor de Fibonacci.
- Al final, se devuelve el valor calculado como un objeto BigInteger.

3.3. Fibonacci con Memorización no Iterativo

Para la implementación con memorización no iterativo de Fibonacci utilizamos un mapa de memorización que nos servirá para almacenar los números de Fibonacci que ya han sido calculados, lo que ayuda a evitar cálculos redundantes y mejora la eficiencia del algoritmo. Este enfoque es más eficiente que el enfoque puramente recursivo para calcular los números de Fibonacci, especialmente para valores más grandes de n, porque evita recalcular los mismos números de Fibonacci varias veces.

Listing 3: Fib_Memorizacion.java

```
import java.math.BigInteger;
import java.util.HashMap;
import java.util.Map;

public class Fib_Memorizacion_no_Iterativo {
    private static Map<Integer, BigInteger> memo = new HashMap<>();
```

```
7
8       public static void main(String[] args) {
9           int x = (args.length > 0) ? Integer.parseInt(args[0]) : 0;
10
11          long tmpInicio = System.currentTimeMillis();
12
13          BigInteger result = fibonacci_memorizacion(x);
14
15          long tmpFin = System.currentTimeMillis();
16          long tmpTrans = tmpFin - tmpInicio;
17
18          if (args.length > 1 && args[1].equals("true")) {
19              System.out.println("\nFibonacci Memorizacion (" + x + ") = " + result);
20          }
21
22          System.out.println("\nTiempo de ejecucion: " + tmpTrans + " milisegundos\n\n");
23      }
24
25      public static BigInteger fibonacci_memorizacion(int n) {
26          if (n <= 1) {
27              return BigInteger.valueOf(n);
28          } else if (memo.containsKey(n)) {
29              return memo.get(n);
30          } else {
31              BigInteger result = fibonacci_memorizacion(n - 1).add(fibonacci_memorizacion(n -
                    2));
32              memo.put(n, result);
33              return result;
34          }
35      }
36  }
```

Explicación de la función fibonacci_memorizacion

- Es una función que calcula el número de Fibonacci para un índice dado n utilizando el enfoque de memorización (memoization).
- La primera parte de la función. Verifica si n es menor o igual a 1. Si es así, significa que n es un caso base de la secuencia de Fibonacci, y simplemente devuelve n convertido en un objeto `BigInteger` utilizando `BigInteger.valueOf`(n). En este caso, el resultado será 0 para $n = 0$ y 1 para $n = 1$.
- La segunda parte de la función. Si n no es un caso base (es decir, n es mayor que 1), verifica si el resultado para n ya está almacenado en el mapa de memorización (`memo`). Si el resultado ya está en el mapa, lo recupera utilizando `memo.get`(n) y lo devuelve. Esto evita calcular el mismo número de Fibonacci más de una vez y mejora la eficiencia del algoritmo.
- La tercera parte de la función. Si n no es un caso base y el resultado no está en el mapa de memorización, se calcula el número de Fibonacci para n de manera recursiva. Esto se hace llamando a `fibonacci_memorizacion`(n - 1) para obtener el número de Fibonacci para $n - 1$ y `fibonacci_memorizacion`(n - 2) para obtener el número de Fibonacci para $n - 2$. Luego, se suman estos dos resultados para obtener el número de Fibonacci para n. El resultado se almacena en el mapa de memorización (`memo.put`(n, result)) para futuras referencias y se devuelve como un objeto `BigInteger`.

3.4. Fibonacci Logaritmico

Para la implementación logarítmica de Fibonacci se utiliza la técnica de .exponenciación rápida". Esta estrategia eficiente permite calcular los números de Fibonacci en tiempo logarítmico en lugar de un enfoque lineal. La clave de esta eficiencia reside en el uso de matrices y la reducción de operaciones redundantes. Además, esta técnica es especialmente beneficiosa cuando se trabaja con números grandes de la secuencia de Fibonacci, ya que evita el desbordamiento de valores y mejora significativamente el rendimiento en comparación con la implementación recursiva o iterativa estándar.

Listing 4: Fib_logaritmico.java

```
1
2  import java.math.BigInteger;
3
4  public class Fib_logaritmico {
5
6      public static void main(String[] args) {
7        int x = (args.length > 0) ? Integer.parseInt(args[0]) : 0;
8
9        long tmpInicio = System.currentTimeMillis();
10
11       BigInteger result = fibonacci_logaritmico(x);
12
13       long tmpFin = System.currentTimeMillis();
14       long tmpTrans = tmpFin - tmpInicio;
15
16       if (args.length > 1 && args[1].equals("true")){
17         System.out.println("\nFibonacci Logaritmico (" + x + ") = " + result);
18       }
19
20       System.out.println("\nTiempo de ejecucion: " + tmpTrans + " milisegundos\n\n");
21     }
22
23     public static BigInteger fibonacci_logaritmico(int n) {
24         if (n <= 0) {
25             return BigInteger.ZERO;
26         }
27
28         BigInteger[][] matriz = {{BigInteger.ONE, BigInteger.ONE}, {BigInteger.ONE,
               BigInteger.ZERO}}; //{0, 0} {1, 1}
29         elevarMatriz(matriz, n - 1);
30
31         return matriz[0][0];
32     }
33
34     public static void multiplicarMatriz(BigInteger[][] A, BigInteger[][] B) {
35         BigInteger a = A[0][0].multiply(B[0][0]).add(A[0][1].multiply(B[1][0]));
36         BigInteger b = A[0][0].multiply(B[0][1]).add(A[0][1].multiply(B[1][1]));
37         BigInteger c = A[1][0].multiply(B[0][0]).add(A[1][1].multiply(B[1][0]));
38         BigInteger d = A[1][0].multiply(B[0][1]).add(A[1][1].multiply(B[1][1]));
39
40         A[0][0] = a; // a b
41         A[0][1] = b; // c d
42         A[1][0] = c;
43         A[1][1] = d;
44     }
45
```

```
46      public static void elevarMatriz(BigInteger[][] matriz, int n) {
47          if (n <= 1) {
48              return;
49          }
50
51          elevarMatriz(matriz, n / 2);
52          multiplicarMatriz(matriz, matriz);
53
54          BigInteger[][] base = {{BigInteger.ONE, BigInteger.ONE}, {BigInteger.ONE,
                BigInteger.ZERO}};
55          if (n % 2 != 0) {
56              multiplicarMatriz(matriz, base);
57          }
58      }
59  }
```

Explicación de la función fibonacci_logaritmico

- La función fibonacci_logaritmico calcula el valor de Fibonacci de manera logarítmica utilizando la técnica de exponentiación rápida.
- Si n es menor o igual a 0, se devuelve 0 como un objeto BigInteger (caso base).
- Se crea una matriz matriz inicializada con valores iniciales 1, 1, 1, 0, que se utiliza para realizar la exponentiación rápida.
- Se llama a la función elevarMatriz para elevar la matriz a la potencia n - 1. El resultado se encuentra en matriz[0][0] y representa el valor de Fibonacci deseado.

Explicación de la función multiplicarMatriz

- La función multiplicarMatriz implementa la multiplicación de dos matrices cuadradas A y B y actualiza la matriz A con el resultado. Este es un paso fundamental en la técnica de exponentiación rápida.

Explicación de la función elevar_matriz

- La función elevarMatriz implementa la técnica de exponentiación rápida para elevar la matriz matriz a la potencia n.
- La recursión se utiliza para dividir el problema en subproblemas más pequeños.
- Se llama a elevarMatriz(matriz, n / 2) para calcular matriz$\hat{}$(n/2) y luego se multiplica matriz por sí misma utilizando multiplicarMatriz(matriz, matriz).
- Si n es impar (n % 2 != 0), se multiplica matriz por base para tener en cuenta el término adicional.

3.5. Pruebas de Eficiencia

3.5.1. Recursiva

```
User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ javac Fib_Recursivo.java

User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Recursivo.java 20

Tiempo de ejecucion: 2 milisegundos



User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Recursivo.java 30

Tiempo de ejecucion: 22 milisegundos



User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Recursivo.java 50

Tiempo de ejecucion: 117215 milisegundos
```

Como pueden ver el tiempo de ejecución es bastante para tan solo 50 números, por lo tanto no es recomendable utilizar el método recursivo, ya que al momento de su ejecución es muy redundante.

3.5.2. Iterativo

```
> java .\Fib_Iterativo.java 1000

Tiempo de ejecucion: 1 milisegundos

> java .\Fib_Iterativo.java 10000

Tiempo de ejecucion: 5 milisegundos

> java .\Fib_Iterativo.java 100000

Tiempo de ejecucion: 156 milisegundos

> java .\Fib_Iterativo.java 1000000

Tiempo de ejecucion: 12033 milisegundos

> java .\Fib_Iterativo.java 10000000

Tiempo de ejecucion: 1101598 milisegundos
```

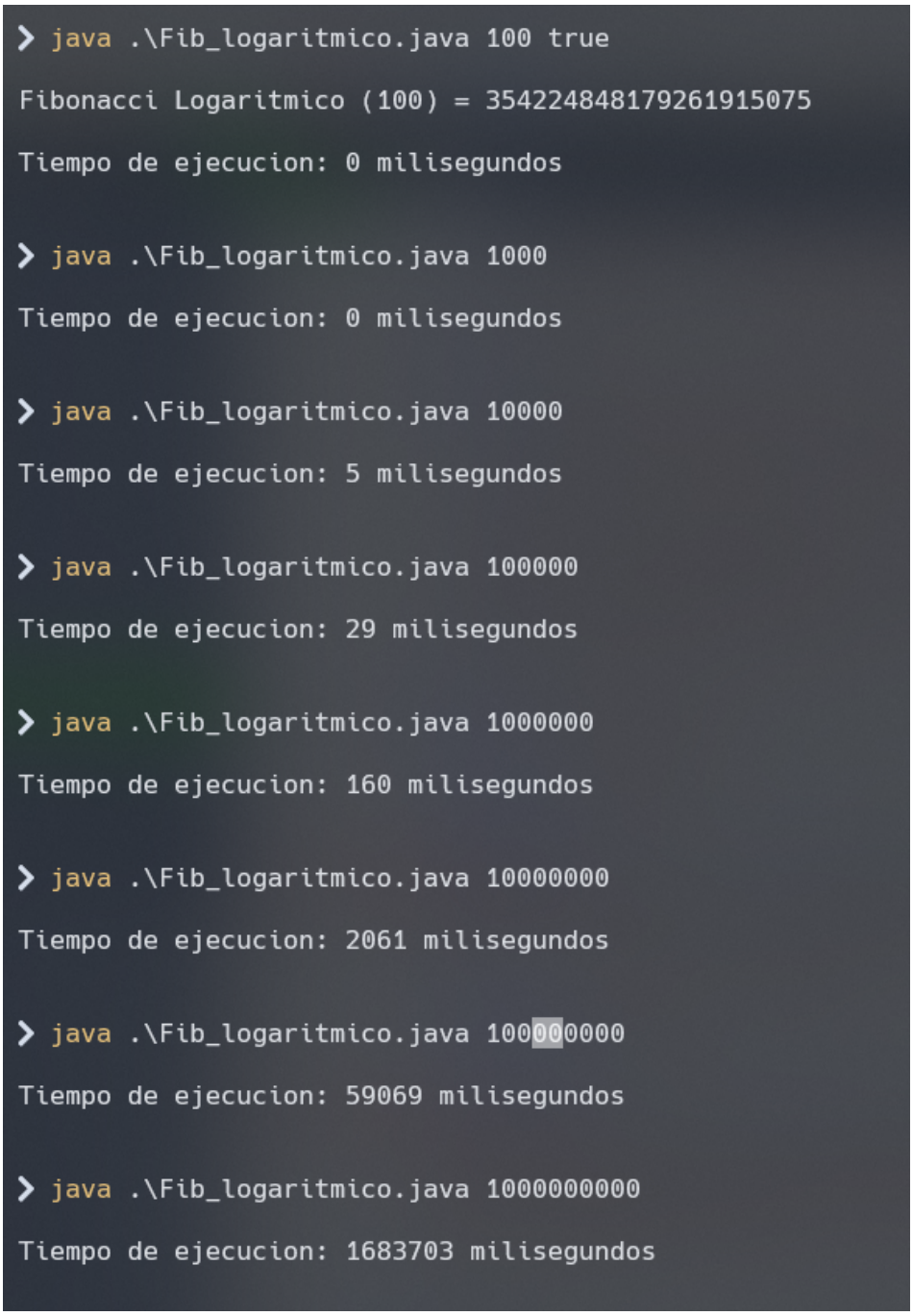
Como se puede observar en la anterior imagen, para calcular el número 10 millones en la serie de Fibonacci se demoró 18 minutos aprox.

3.5.3. Memorización no iterativa

```
User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Memorizacion_no_Iterativo.java 100

Tiempo de ejecucion: 1 milisegundos



User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Memorizacion_no_Iterativo.java 1000

Tiempo de ejecucion: 1 milisegundos



User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java Fib_Memorizacion_no_Iterativo.java 1500

Tiempo de ejecucion: 2 milisegundos



User@DESKTOP-7OUENGD MINGW64 ~/eda-lab2 (main)
$ java -Xss4m Fib_Memorizacion_no_Iterativo 10000

Tiempo de ejecucion: 10 milisegundos
```

Es más eficiente que el recursivo de por si ya que este almacena valores y omite calculos ya redundantes, pero aún así no es la mejor alternativa.

3.5.4. Logaritmica



Como se puede observar en la anterior imagen, este método es muy eficiente, el mejor de los que se presentaron, pudiendo calcular el número **1 BILLÓN** en un tiempo de 28 minutos aprox. Mientras que el número 100 MILLONES en un tiempo de 1 minutos.

4. Conclusiones

- El enfoque recursivo es útil para comprender la naturaleza de la secuencia de Fibonacci, pero no es eficiente para cálculos prácticos de números grandes. Por otro lado, el enfoque con memorización es altamente eficiente y es la opción preferida cuando se requieren cálculos precisos y rápidos de la secuencia de Fibonacci, incluso para números grandes.
- La implementación de la secuencia de Fibonacci utilizando la técnica de .[ex]ponentiación rápida.[es] una estrategia altamente eficiente, especialmente cuando se trabaja con números grandes. Este enfoque logarítmico reduce significativamente el tiempo de cálculo y evita problemas de desbordamiento de valores, lo que lo convierte en una opción poderosa para calcular la secuencia de Fibonacci en escenarios donde la eficiencia es esencial.

5. Referencias

- `https://www.cs.us.es/~jalonso/cursos/i1m-15/temas/tema-24.html`
- `https://dev.java/learn/`
- `https://www.ugr.es/~eaznar/fibo.htm`